Decomposition and Diet Problems

Danny Hamilton

Doctor of Philosophy
University of Edinburgh
2009

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Danny Hamilton*)

I'd like to thank my supervisor Ken McKinnon and my second supervisor Julian Hall. I'd also like to thank Format International, for their support and for providing many interesting problems.

Abstract

The purpose of this thesis is to efficiently solve real life problems. We study LPs. We study an NLP and an MINLP based on what is known as the *generalised pooling problem* (GPP), and we study an MIP that we call the *cattle mating problem*. These problems are often very large or otherwise difficult to solve by direct methods, and are best solved by decomposition methods. During the thesis we introduce algorithms that exploit the structure of the problems to decompose them.

We are able to solve row-linked, column-linked and general LPs efficiently by modifying the tableau simplex method, and suggest how this work could be applied to the revised simplex method.

We modify an existing sequential linear programming solver that is currently used by Format International to solve GPPs, and show the modified solver takes less time and is at least as likely to find the global minimum as the old solver. We solve multifactory versions of the GPP by augmented Lagrangian decomposition, and show this is more efficient than solving the problems directly. We introduce a decomposition algorithm to solve a MINLP version of the GPP by decomposing it into NLP and ILP subproblems. This is able to solve large problems that could not be solved directly. We introduce an efficient decomposition algorithm to solve the MIP cattle mating problem, which has been adopted for use by the Irish Cattle Breeding Federation.

Most of the solve methods we introduce are designed only to find local minima. However, for the multifactory version of the GPP we introduce two methods that give a good chance of finding the global minimum, both of which succeed in finding the global minimum on test problems.

Contents

Chapter 1

Introduction

1.1 Overview

In this thesis we solve optimisation problems. These are problems with an objective function, to be maximised or minimised, and a set of constraints on the decision variables. The nature of the constraints of a problems determines its structure. It has long been known that the structure of a problem is important, and that methods that exploit the structure of a problem can solve it efficiently.

The first optimisation problems solved were the linear programs (LPs) introduced by Dantzig [1947], and solved by his tableau simplex method. This was followed by decomposition methods that allow a large LP, which is difficult or even impossible to solve directly, to be broken down into several smaller problems. These methods include Dantzig-Wolfe decomposition, Dantzig and Wolfe [1960], (and its dual, Lagrangian Relaxation) and Benders decomposition, Benders [1962]. The introduction of the revised simplex method allowed faster solves of LPs, particularly for problems with sparse constraints, which are common in practice. Hall and McKinnon [2005] showed how sparsity can be exploited in the revised simplex method. With modern computers problems can be solved a lot more quickly than in the days of Dantzig, but as computer speed increases so does the size of the problems that are required to be solved, and so there is always a need for efficient solvers.

Variants of LP have also been introduced. The addition of nonlinear constraints makes LPs into nonlinear programs (NLPs), the addition of integer constraints makes LPs into mixed integer programs (MIPs), and the addition of both at once makes LPs into mixed integer nonlinear programs (MINLPs). All of these new problem types are nonlinear, and can have multiple local minima. For each problem type special methods have been developed to solve them. Very often these methods involve reducing a problem to an easier type. For example, the method of sequential linear programming (SLP) solves NLPs by solving a series of LPs, and similarly the method of branch and bound solves MIPs by solving a series of LPs. The

advantage of such methods is that a hard problem can be reduced to a series of smaller problems, much like the decomposition algorithms of Dantzig Wolfe and Benders. Indeed a MINLP can be decomposed into a series of NLPs and MIPs, and the NLPs and MIPs each solved by solving a series of LPs. Thus decomposition methods are powerful tools for solving difficult problems. Just as with direct methods, some decomposition methods attempt to find the global minimum, others simply try and find a good local minimum.

Optimisation problems occur naturally in many real world situations. In particular, there are *diet problems*, which involve mixing ingredients to form a cheap but nutritious final product. The simplest of these is known as the *blending problem*, and can be modelled as an LP. Blending problems are routinely solved by linear programming in factories around the world. In practice factories often have certain extra features that lead to problems which are much harder to solve than the blending problem, such as the *pooling problem*. This is a blending problem with the addition of mixing bins.

Another common optimisation problem is the *assignment problem*, where objects from one set must be assigned to objects in another set, at minimum total cost. In its simplest form this problem can be modelled as an LP. With compliciations to the model it can become significantly harder to solve. In this thesis we develop new decomposition methods and improve existing methods to solve such problems.

As the work in this thesis covers several areas, within each chapter we introduce that topic. At the end of each chapter we also give a summary, and, where appropriate, suggestions for further work.

1.2 Application

The work in this thesis is intended to be of practical benefit. In particular we have worked in conjunction with several companies.

Format International currently use University of Edinburgh software to solve diet problems on behalf of clients, and have an interest in solving further problems. They are an *independent software company specialising in recipe optimization, ingredient allocation and food and feed formulation solutions for animal feed, pet food, human food, and other industries*. For Agricultural and aqua feeds *Format's programs are the most widely used advanced optimization, ingredient and resource management software in this sector, responsible for the manufacture of more tonnes on a global basis that any other provider* (Format International [2009]).

Format International's clients include Sloten International, who are based in the Netherlands. Sloten *develop, manufacture and market special feeds for young animals* (Sloten International [2009]).

The Irish Cattle Breeding Federation (ICBF) is an organisation whose intention is *to achieve the greatest possible genetic improvement in the national cattle herd for the benefit of Irish*

Farmers (ICBF [2006]).

1.3 Contributions of the thesis

In Chapter 2 and Chapter 3 we study LPs. In Chapter 2 we explain the occurrence of a phenomenon known as *hyper-sparsity*, which makes some LPs easy to solve. We use novel tests to assess the impact of numerical and structural cancellation on LPs, and confirm the widely held opinion that the most important attribute for determining the difficulty of solving a problem is its structure. A new theorem is given to explain the limited increase in density of block structured row and column linked problems. In Chapter 3 the knowledge from Chapter 2 is used to motivate several new modifications to the standard simplex method, that solve *row linked* and *column linked* problems efficiently. The best of these algorithms is shown to be effective on a large set of general LPs, and is also adapted for use with the revised simplex method.

In Chapter 4 we describe the *generalised pooling problem* (GPP), an NLP that is an extension of the *pooling problem*. This is a problem that is currently solved by Format International. We give a new formulation of the model and offer fresh insight into how the mixing bins cause nonlinearity in pooling problems. In Chapter 5 we find tight linear relaxations to the GPP. We solve a test set of GPPs using SLP, a method which we confirm as being effective for pooling problems, using an improved SLP solver. We then offer a new solution to the problem of how many random start solves to do before being confident of having found the global minimum.

In Chapter 6 several GPPs are combined to form what we call the *multifactory problem*, which is a large, but highly structured, NLP. This models the situation of one company who own several factories, all of which are supplied from the same limited supply of raw materials. This is a new problem Format International envision solving in the future. We exploit the structure of this problem to solve it efficiently using a novel version of *augmented Lagrangian* decomposition. This is shown to take less estimated work than a direct solve of the undecomposed problem. We also introduce two new algorithms to find the global minimum of the multifactory problem, exploiting the fact that the constituent factories of a multifactory problem are nearly separable.

In Chapter 7 we model an unstudied problem from Sloten International, of making milk replacement fluid to be fed to young animals. This is done in a factory that can be modelled by a GPP, with the additional feature of *silos* and *weighing belts*, which make the problem a MINLP. We introduce a new algorithm to solve such MINLPs by decomposing them into NLP and MIP subproblems.

In Chapter 8 we model the ICBF *cattle mating problem*, which is the problem of assigning which bulls should be selected to mate with which cows in a farmer's herd. This is an MIP that we solve with a novel decomposition.

1.4 Notation

A lot of notation is needed to introduce the different types of problem. Throughout the thesis we tend to use lower case letters to designate variables, and upper case letters to designate parameters. An exception is when defining a standard LP, where we use the conventional notation of $\boldsymbol{c}$ and $\boldsymbol{b}$ being vectors of parameters. Occasionally it is convenient to designate bounds on a variable by using bars above and below the variable name, for example bounds bounds on y_{mn} of $\underline{y}_{mn} \leq y_{mn} \leq \overline{y}_{mn}$. In the description of the diet problems Greek letters are always used for variables measuring flow in the network.

The use of bold font ($\boldsymbol{h}$ instead of h), indicates a vector of values. If dimensions for such a vector are not given it can be assumed to be of appropriate size. When we wish to refer to the fixed value of a variable, for example the most recent value of t_1, this is designated by $\hat{t}_1$. The optimum value of a variable, such as t_1 is designated by t_1^*. We use $\boldsymbol{x}$ for a general variable. An altered version of a variable or parameter that has already been defined is marked with a dash, for example a basis matrix $\boldsymbol{B}$ may be altered to become $\boldsymbol{B'}$.

We use the term *method* to denote an algorithm, for example the method of Benders decomposition. A *solve* is one solution of each problem, for example Solve 1 could be a solve of each of a given set of 10 problems with the method of Benders decomposition. A *run* is a series of repeated solves of the same problem, for example Run 1 could be 50 different solves of a given NLP, starting from different points.

Chapter 2

Sparsity in the Simplex Method

In both Chapter 2 and Chapter 3 we study LPs. Chapter 2 serves as background for Chapter 3. In Section 2.1 we introduce some key concepts. In Sections 2.2 and 2.3 we investigate properties of problems that affect their difficulty to solve. In Section 2.2 we use novel tests to assess the impact of numerical and structural cancellation on LPs, and confirm the widely held opinion that the most important attribute for determining the difficulty of solving a problem is its structure. In Section 2.3 we study basis matrices in *Tarjan form*. We present some new analysis of the link between the basis inverse and he tableau density, including a novel way to view the nonbasis, and two simple observations relating to the basis inverse. Theorem 2 is given to explain the limited increase in density of block structured row and column linked problems.

2.1 Introduction

Here we introduce linear programs, the simplex method, the Subnet problems and the concept of *Hyper-Sparsity*.

2.1.1 Linear programs

A linear program (LP) is a problem that can be expressed in the form:

$$\begin{aligned} \min_{\boldsymbol{x}} \; & f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x}, \\ \text{s.t. } & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \; \in \; \boldsymbol{X}, \end{aligned} \tag{2.1}$$

where $\boldsymbol{x}$ is a vector of n continuous decision variables, $\boldsymbol{c}$ is a vector of n costs, $\boldsymbol{A}$ is an m by n matrix, $\boldsymbol{b}$ is a vector of size m, and $\boldsymbol{X}$ is a set of n lower and upper bounds on $\boldsymbol{x}$. The program

is said to be linear, as the objective and all the constraints are linear functions of $\boldsymbol{x}$, meaning their graph in a Cartesian coordinate plane (of appropriate dimensions) is a straight line.

In order to make the inequalities into equalities (2.1) may be augmented with the addition of extra variables:

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{z}} \; & f(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t. } & \boldsymbol{A}\boldsymbol{x} + \boldsymbol{I}\boldsymbol{z} = \boldsymbol{b}, \\ & \boldsymbol{x} \in \boldsymbol{X}, \\ & \boldsymbol{z} \geq \boldsymbol{0}, \end{aligned} \tag{2.2}$$

where $\boldsymbol{z}$ is a vector of m further continuous decision variables and $\boldsymbol{I}$ an identity matrix of size m. Let $\boldsymbol{x'}$ be the concatenation of $\boldsymbol{x}$ and $\boldsymbol{z}$, $\boldsymbol{c'}$ be the concatenation of $\boldsymbol{c}$ and a vector of zeroes, $\boldsymbol{A'}$ be the concatenation of $\boldsymbol{A}$ and an identity matrix, and $\boldsymbol{X'}$ be the concatenation of $\boldsymbol{X}$ and bounds on the $\boldsymbol{z}$ variables. Then (2.2) may be rewritten as:

$$\begin{aligned} \min_{\boldsymbol{x'}} \; & f(\boldsymbol{x'}) = \boldsymbol{c'}^T\boldsymbol{x'}, \\ \text{s.t. } & \boldsymbol{A'}\boldsymbol{x'} = \boldsymbol{b}, \\ & \boldsymbol{x'} \in \boldsymbol{X'}. \end{aligned} \tag{2.3}$$

We have now returned to a form that is almost identical to (2.1), but with equality instead of inequality constraints and a larger vector of variables $\boldsymbol{x'}$. We have shown that an LP with inequalities may be rewritten as one with equalities only. Most LPs have data in the form (2.1), but are converted to the form (2.3) to be solved.

2.1.2 Simplex method

We will solve LPs using the *simplex method*, introduced by Dantzig in 1947. We give some definitions relevant the the simplex method. There are two main implementations of the simplex method, the *tableau simplex method* and the *revised simplex method*. The definitions introduced below are general to both implementations.

Consider the LP presented in (2.1). In the simplex method the variables $\boldsymbol{x}$ are known as *logical variables* or *logicals*, and the matrix $\boldsymbol{A}$ as the *tableau*. During the solve of an LP by the simplex method the tableau may be manipulated by row and column operations. The resulting matrices that occur are still referred to as the tableau, with the original matrix $\boldsymbol{A}$ referred to as the *initial tableau*.

We call the variables $\boldsymbol{z}$ from (2.2) *structural variables* or *structurals*. We call the matrix $\boldsymbol{A'}$

from (2.3) the *initial extended tableau*. Versions of $\boldsymbol{A'}$ that may appear during a solve are all still known as the extended tableau. The extended tableau always contains a (possibly permuted) copy of the identity matrix, but is otherwise the same as the tableau.

The simplex method divides the columns of the initial extended tableau into a *basic* set, which collectively form a square matrix known as the *basis*, and a *nonbasic* set, which collectively form as matrix that we call the *nonbasis*. The nonbasic variables are always at a lower or upper bound, this is not so for the basic variables. At iteration k we refer to the basis and nonbasis as $\boldsymbol{B_k}$ and $\boldsymbol{N_k}$ respectively. The extended tableau can be written in terms of the basis and nonbasis, using the relationship $\boldsymbol{A'_k} = \boldsymbol{B_k}^{-1}\boldsymbol{N_k}$.

In each iteration of the simplex method one column moves from the nonbasis to the basis, and one column moves from the basis to the nonbasis. The iterations continue until the optimal basis is found and the problem is therefore solved. As the basis changes the initial costs $\boldsymbol{c}$ are updated to form the *reduced costs*, which indicate the rate of change of the objective value at each basis. The right hand side vector $\boldsymbol{b}$ is also updated each iteration, as it the objective value.

Any value of the variables that satisfies all the constraints is said to be a *feasible point*. The feasible point with best objective value is said to be *optimal*. During a solve we may encounter points where some constraints are violated. The number of violated constraints is referred to as the *number of infeasibilities*, and the total violation of all constraints is referred to as the *sum of infeasibilities*. The iterations of the simplex method can be broken down into two phases. In *phase one* we minimise the sum of infeasibilities to find a feasible point. In *phase two* we retain feasibility and minimise the objective value.

A tableau with few nonzeros in it is said to have a low density, and be *sparse*. Informally, a problem with a sparse initial tableau may be referred to as a sparse problem. Many problems begin sparse then become dense. We refer to the increase in nonzeros in the tableau as *fill in*. The density of the tableau affects the density of the pivot column and row. As pivoting copies the pivot column and row across the tableau the density of the pivot row and column affects the amount of work required. It is therefore desirable to preserve sparsity of the tableau during the solve. Variations of the simplex method which exploit sparsity can give orders of magnitude improvements in solve time (see e.g. Hall and McKinnon [2005]).

The way each iteration is performed varies between the two main implementations of the simplex method, called the tableau simplex method and the revised simplex method. The tableau simplex method is simple to implement and the algorithm is easily manipulated, so is what we use in this thesis. However the revised simplex method is much faster, so the purpose of these chapters is to use the tableau simplex method to understand why some problems solve easily, and use that knowledge to improve the revised simplex method.

Tableau simplex method

Our solver, Simplex10, uses the *tableau simplex method*. The tableau simplex algorithm is well described in e.g. Winston [2000]. An outline is given in Table 2.1.

1	Scan the reduced costs to find an attractive column. Call this the pivot column.
2	Compare the pivot column and right hand side vector to find the row limiting movement of the pivot column. Call this the pivot row.
3	Using the pivot column and pivot row update the tableau, reduced costs, right hand sides and objective value by pivoting.

Table 2.1: An iteration of the tableau simplex method

In Step 1 we scan of the reduced costs, to find the most attractive column to increase or decrease. Note that *variables* correspond to *columns* of the extended tableau, and the terms can be used somewhat interchangeably. The chosen column is designated as the *pivot column*. In fact it is not necessary to choose the very best column each iteration, and this freedom in selecting the pivot column allows different versions of the algorithm.

In Step 2 the pivot column is compared to the vector of right hand sides, to indicate the maximum feasible movement in the direction of the pivot column. The row which limits the movement is known as the *pivot row*, and the element of the tableau at the intersection of pivot column and pivot row is the *pivot element*. In fact it is possible to select a row other than the one suggested, to allow some infeasibility, leading to several different possible rules for selecting the pivot row.

In Step 3 the pivot column enters the basis, and a column determined by the pivot row leaves the basis, in a process called *pivoting*. This alters the tableau by copying the pivot column and row across the tableau. Note that the whole table is explicitly stored and updated in the tableau simplex method. Iterations continue, as long as there are columns that are attractive to move.

The description above assumes the initial point to the problem is feasible. This is called a *phase two* problem. If the initial point is not feasible then an enlarged LP can be formed and solved to find a feasible point. This enlarged *phase one* LP can be solved in the same way as the phase two LP. In phase one (and in phase two) we may allow greater flexibility in the algorithm by picking pivot columns that are good for some given combination of objective value and infeasibility, and picking pivot rows that allow the problem to become (more) infeasible.

The *Markowitz count* measured each iteration is the product of the number of nonzeros in the pivot row, excluding the pivot element itself, with the number of nonzeros in the pivot column, again excluding the pivot element. A good implementation of the tableau simplex method only performs arithmetic operations on nonzero tableau values and for such an implementation an approximate measure of the work required to solve a problem is the sum of the Markowitz counts. In this thesis we measure the work using a slightly adjusted Markowitz count, which is simply the product of the number of nonzeros in the pivot row and pivot column. This

adjustment reflects the fact that as well as work being required each iteration to perform the pivot, work is also required on the pivot row and column themselves.

Simplex10

We stated above that the tableau simplex method can be used with different row and column selection rules. In this chapter we solve LPs using a tableau simplex solver called Simplex10, written by the author. We now briefly describe the default pivot column and pivot row selection rules used by Simplex10. In later sections these rules will be varied as we explore different methods to solve LPs.

In Simplex 10 the pivot column is selected by *steepest edge pricing*. This gives each nonbasic column i that is free to move in the appropriate direction a score s_i, calculated from the *reduced cost* of the column divided by the size of the column. In our implementation the size of a column is calculated by the Euclidean norm. The column i with the largest s_i becomes the next pivot column.

Pivot row selection is more complicated. Once the pivot column is selected there are a limited number of rows of the tableau that intersect that column. Among these rows there can be *bound swap rows*, where pivoting on that row constitutes simply moving the pivot column from one bound to another, and *equality rows*. When selecting the pivot row we consider only those candidate rows that do not increase the sum nor the number of infeasibilities. A bound swap row that strictly improves the sum of infeasibilities is automatically chosen. If none exists, rows that lead to sums of infeasibilities close to the minimum available or are equality rows are considered, and from these the row with the largest pivot element is chosen.

We also note that Simplex 10 begins with all logical variables basic, and all structural variables nonbasic.

Revised simplex method

Nowadays the *revised simplex method* is usually preferred to the tableau simplex method. In the revised simplex method the reduced costs and right hand sides are stored and updated as in the tableau simplex method, but the tableau is not explicitly stored. Instead only the basis inverse is stored (indirectly, as a product of several matrices), as well as the original nonbasis $\boldsymbol{N}$. The pivot column and pivot row are generated sparsely as needed. This makes the revised simplex method more efficient (and numerically stable) than the tableau simplex method for large sparse problems, which are common in practice. An outline of each iteration of the revised simplex method is given in Table 2.2, adapted from Hall and McKinnon [2005].

For the revised simplex method an appropriate measure of work is the sum of the nonzeros in the pivot row and column.

1	Scan the reduced costs to find an attractive column, by CHUZC operation.
2	Form this pivot column, by FTRAN operation.
3	Compare the pivot column and right hand side vector to find the row limiting movement of the pivot column, by CHUZR operation.
4	Form this pivot row, by BTRAN and PRICE operations.
5	Update the representation of the basis, by UPDATE operation.
6	Update the reduced costs, right hand sides and objective value.

Table 2.2: An iteration of the revised simplex method

2.1.3 Subnet problems

In this chapter we explore properties of problems belonging to a new set of LPs we call the Subnet problems. Although this is a new test set it consists of known problems taken from other test sets. It contains many problems from the Netlib problem set with a few from the Mittelmann and Kennington problem sets. The resulting set of 73 problems are mostly LPs that have been derived from various real world problems. The problems range from a few dozen rows and columns to thousands of rows and columns. The solve times, by the tableau simplex method, range from less than a second to several minutes.

Because some of the Subnet problems are fairly large (to be solved with a tableau simplex solver), not all of them can be solved by all of the methods that we use. Therefore in the random simulation solves of Sections 2.2.2 and 2.2.3, and the work requiring what we call the *Tarjan form* in Sections 3.4.4 and 3.4.5, we omit some problems from Subnet.

2.1.4 Hyper-Sparsity

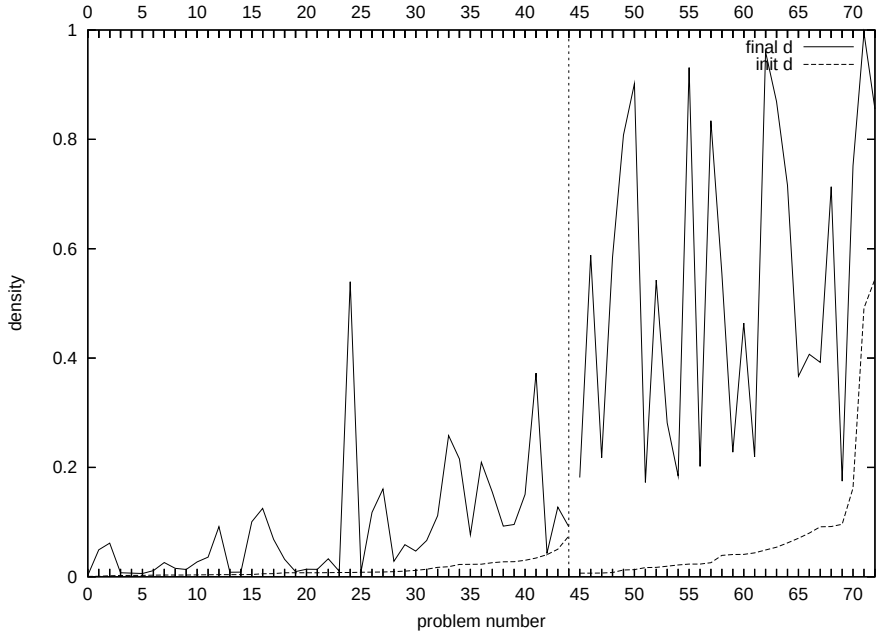
Hyper-Sparsity is a property of some LPs. We categorise each of the LPs in Subnet as Hyper-sparse or not, using the definition found in Hall and McKinnon [2005]. Hall calls a problem Hyper-sparse if at least one of the FTRAN, BTRAN and PRICE operations of the revised simplex method has no more than 10% nonzeros on at least 60% of the iterations. Any problems that are not Hyper-sparse we classify as *Densefill*. In terms of the tableau simplex method, Hyper-sparse problems are typically those that begin and finish sparse.

In this work the classification of each problem as Hyper-sparse or Densefill is for presentation purposes only, and we apply the same analysis and solution methods to all problems. Tables A.1 and A.2 in Section A of the appendix section list the Hyper-sparse then the Densefill problems in Subnet.

Table 2.1(a) shows the average density of the initial tableau and average density of the optimal tableau for the Subnet problems. Hyper-sparse problems typically begin and finish with sparser tableau than Densefill problems. In Figure 2.1(b) the problems are split into Hyper-sparse and Densefill then ordered by the density of the initial tableau, within each set. Problems 0-44 are Hyper-sparse, problems 45-72 are Densefill, with a vertical line between

	initial tableau	optimal tableau
Hyper-sparse average	0.013	0.085
Densefill average	0.077	0.539

(a)



(b)

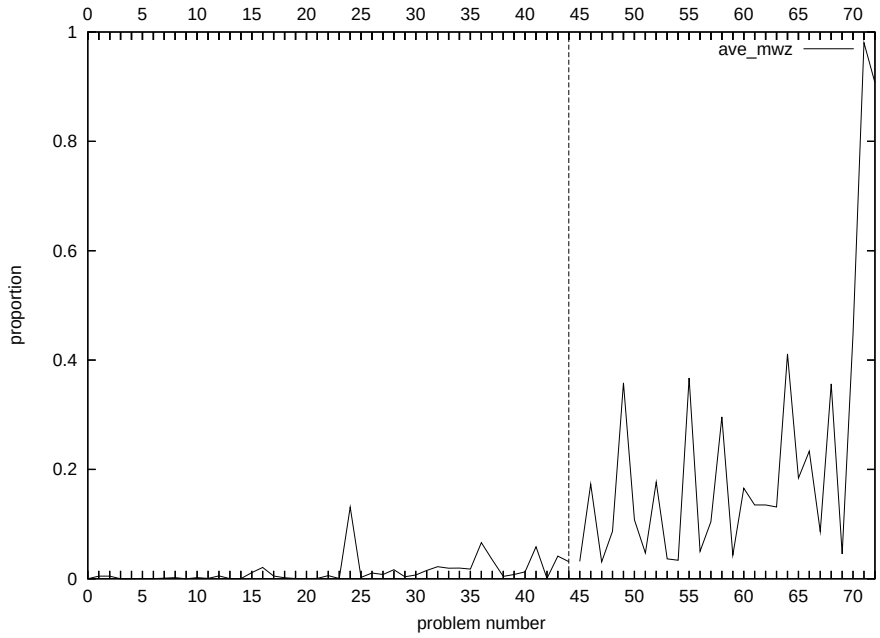
Figure 2.1: Subnet tableau density. Averages in table (a) and values for each problem in graph (b)

them. This problem ordering is used throughout the chapter. The graph shows that there is much variation of initial and final tableau density within each set.

Using our definition of work above, based on the Markowitz counts, if at all iterations each pivot is on a completely dense column and completely dense row the maximum possible work is achieved. Figure 2.2 shows the total work as a proportion of this maximum for each Subnet problem. Hyper-sparse problems on average have a much lower proportion of maximum possible work than Densefill problems.

Hyper-sparse average	0.013
Densefill average	0.220

(a)



(b)

Figure 2.2: Subnet proportion of maximum possible work in solve process. Averages in table (a) and values for each problem in graph (b)

2.2 Potential reasons for sparse tableau

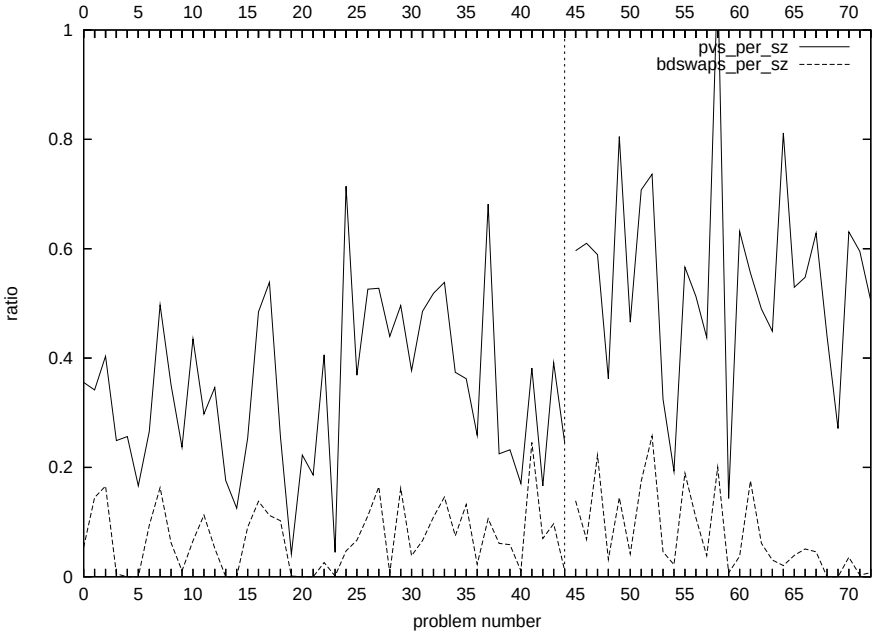
We mentioned above that a sparse tableau leads to sparse pivot rows and columns. This means less work to perform each pivot, hence less total work. Many problems begin sparse but become dense. We now investigate reasons why Hyper-sparse problems stay relatively sparse. There are three reasons identified within Section 2.2; solving in few iterations, *numerical cancellation* and the *structure* of the initial tableau, where structure means the position of the nonzeros. In Section 2.3 we look in more detail at a fourth reason, the structure of the basis. The analysis in these sections confirms the widely held view that it is the structure of the initial tableau of LPs that is the most important in preserving sparsity.

2.2.1 Solving in few iterations

Recall we begin each solve with an all logical basis. Almost all of the Subnet problems solve in more pivots than there are rows in the problem, so there are therefore potentially enough pivots to change the basis from being all logical to consisting of entirely structural variables,

	pivots	bound swaps
Hyper-sparse average	0.343	0.071
Densefill average	0.544	0.079

(a)



(b)

Figure 2.3: Subnet ratio of number of iterations compared to row-column size. Averages in table (a) and values for each problem in graph (b)

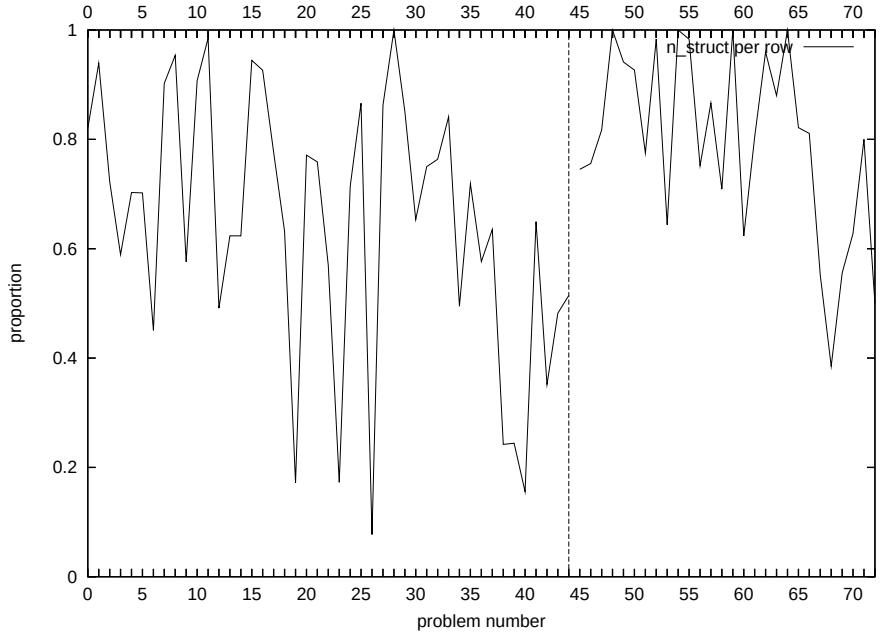
leading to a very dense tableau. However, in practice not all pivots remove a logical and add a structural to the basis. We refer to the number of rows plus the number of columns as the *row-column size* of a problem. Figure 2.3 shows the ratio of number of pivots of each problem to its row-column size, and the ratio of number of bound swaps to row-column size. On average Hyper-sparse problems have fewer pivots and a greater proportion of bound swaps compared to Densefill problems. A few problems are notable. Densefill problem BRANDY (number 58) is the only one to solve in more pivots than its row-column size. The Hyper-sparse STAND problems (numbers 18, 19 and 23) solve in very few iterations. The Hyper-sparse problem STOCFOR1 (number 41) solves with 64% bound swaps.

We mentioned above that an all structural basis leads to an extremely dense tableau. In general the proportion of structurals in the basis correlates with the tableau density. Figure 2.4 shows the proportion of structurals in the optimal basis in the Subnet problems. On average Hyper-sparse problems have a lower proportion of structurals in the optimal basis than Densefill problems.

Collectively the results of this section show that Hyper-sparse problems solve in fewer iter-

Hyper-sparse average	0.698
Densefill average	0.818

(a)



(b)

Figure 2.4: Subnet proportion of structurals in optimal basis. Averages in table (a) and values for each problem in graph (b)

ations, with more bound swaps, and finish with fewer structurals in the basis.

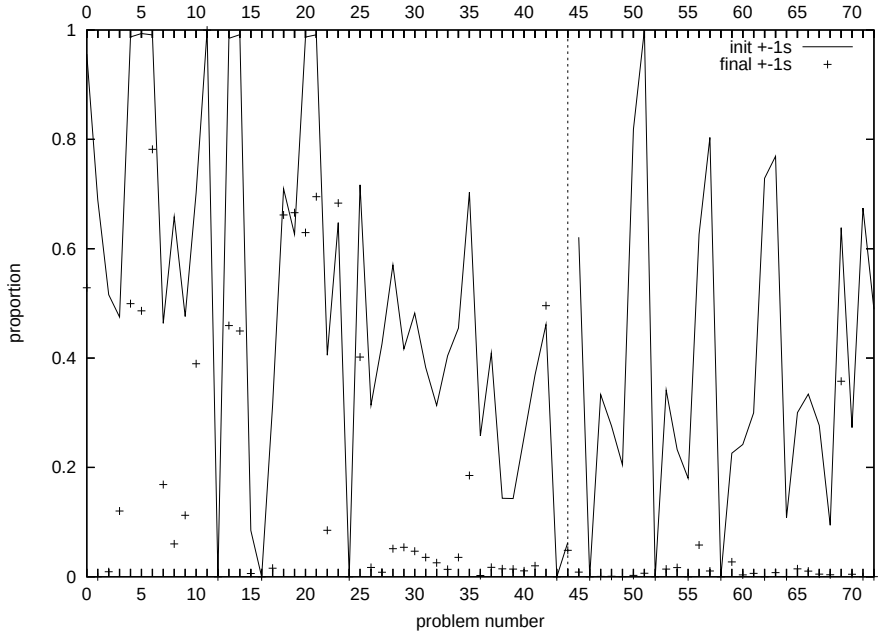
2.2.2 Numerical cancellation

The next reason we consider for some problems staying sparse is numerical cancellation. Numerical cancellation occurs when special values in the pivot rows and columns lead to pivots that cause less fill in than expected. This most commonly occurs due to ± 1s in the initial tableau. Although other values also cause numerical cancellation, ± 1s are significant as they occur naturally in many constraints, and when they are chosen as a pivot element they preserve the size of all other ± 1s in the tableau (but may change their sign). *Network problems* model flow in a network, and have initial tableaux with a large proportion of ± 1s. Several of the Subnet problems are partly or wholly network, such as the Hyper-sparse SHIP problems (numbers 4-5, 13-14, 20-21), SIERRA (number 7), SHELL (number 11) and the Densefill problem DEGEN2 (number 51).

Figure 2.5 shows the proportion of the nonzero values in the initial and optimal tableau of each problem that are ± 1s. On average Hyper-sparse problems have a greater proportion of

	initial tableau	optimal tableau
Hyper-sparse average	0.510	0.222
Densefill average	0.389	0.020

(a)



(b)

Figure 2.5: Subnet proportion of ±1s in tableau. Averages in table (a) and values for each problem in graph (b)

±1s in the initial tableau and in the optimal tableau than Densefill problems. Most problems finish with many fewer ±1s than they started with. An exception is Hyper-sparse problem RECIPE (number 42), where the proportion of ±1s increases due to certain pivots duplicating rows of ±1s.

It is reasonable to wonder how much of an effect the ±1s, and other special numerical values, have in preserving sparsity. To assess this we used a new technique that solved an alternative version of each LP with an initial tableau that had the same sparsity pattern as the original problem, but had random numbers in place of all the original tableau nonzeros. The optimal pivot order from the original solve was then forced on the new tableau. Ten such solves were done for each problem with different sets of random numbers. To avoid numerical difficulties, on the rare occasions that a solve was forced to pivot on a very small pivot element, that solve was aborted and retried with new random numbers. Small tableau values were rounded to zero with the same tolerance as in the original solve. The use of random numbers meant that no numerical cancellation was possible, so the optimal tableaux were at least as dense as in the original solve. The density of the optimal tableau of the random number solves compared to the

density of the optimal tableau in the original solve indicates the effect of numerical cancellation on each problem.

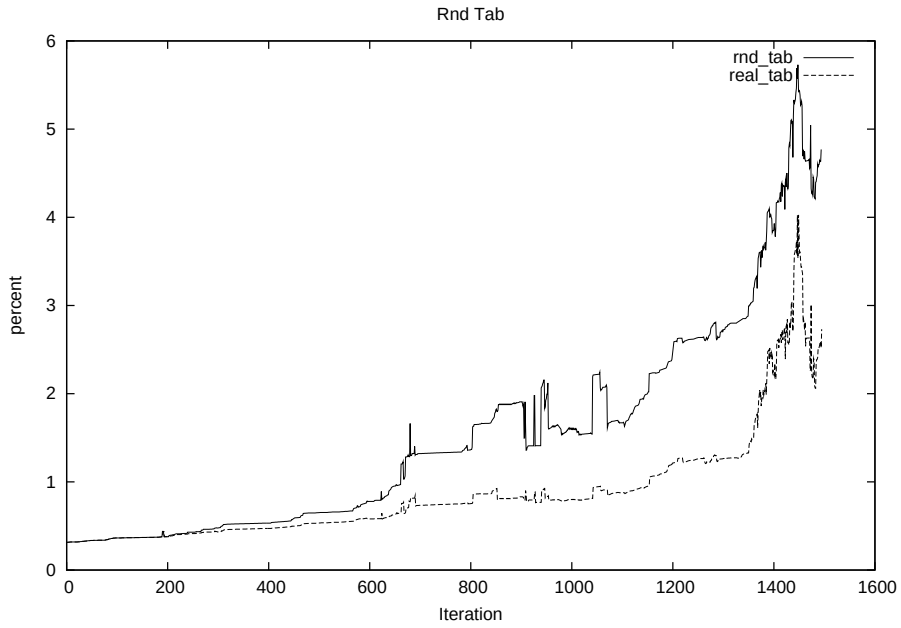


Figure 2.6: Tableau density with random number solve (rnd_tab) and original solve (real_tab)

Figure 2.6 shows the density of the tableau each iteration of the original solve and of a random number solve of Hyper-sparse problem GANGES (number 7). The final density of the original solve is 2.60% compared to 4.26% for the random number solve.

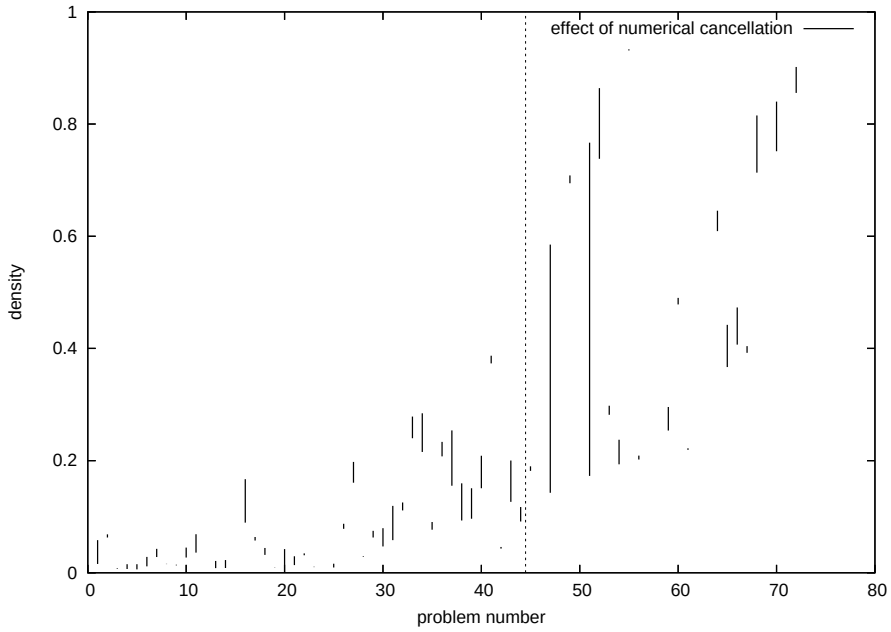
Figure 2.7 shows the effect of numerical cancellation on the Subnet problems. The vertical bars in the graph go from the density of the optimal tableau of the original solve to the average density of the optimal tableau of the random number solves. On average solving with random numbers in the tableau has a larger effect on Hyper-sparse problems, indicating that they benefit more from numerical cancellation. However the most drastic instance of numerical cancellation is in Densefill problem DEGEN2 (number 51), which finishes 17% dense in the original solve and 77% dense in the random number solve.

2.2.3 Tableau structure

The next reason we consider for some problems staying sparse is the structure of the initial tableau, meaning the position of the nonzeros. Certain highly structured problems have been well studied, for example network problems in e.g. Winston [2000], *block structured* problems where the nonzeroes appear mostly within *diagonal blocks* in e.g. Kallio and Porteus [1977], and *staircase* problems where the nonzeros appear mostly within *diagonal bands* in e.g. Fourer [1983]. In this section we analyse the general structure of all the Subnet problems. We first look at the presence of very dense or very sparse columns or rows in the initial tableau, then look at the overall pattern of nonzeros.

Times denser with random number solve	
Hyper-sparse average	1.306
Densefill average	1.103

(a)



(b)

Figure 2.7: Subnet effect of numerical cancellation. Averages in table (a) and values for each problem in graph (b)

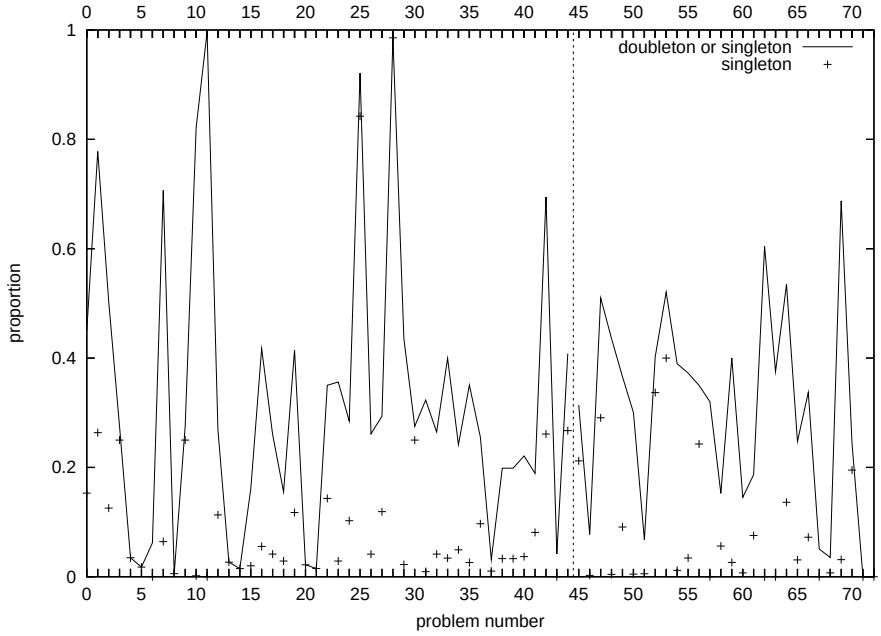
Some problems have *singleton* columns, that contain a single entry, in the initial tableau. These columns never increase the density of the tableau when they are pivoted on. Similarly *doubleton* columns, that contain two entries, can only increase the density slightly when they are pivoted on. Collectively we call singleton and doubleton columns *sparse columns*.

Figure 2.8 shows the proportion of sparse columns in the initial tableau of the Subnet problems. On average Hyper-sparse problems have a greater proportion of sparse columns, though this varies greatly from problem to problem. There are some Hyper-sparse problems with very many sparse columns. One of these, SEBA (number 25), has a large number of singleton columns as the identity matrix is explicitly included in the initial tableau.

Figure 2.9 shows the density of the most dense column and the density of the most dense row of the initial tableaux. A particularly dense row or column in the initial tableau can cause a large increase in the tableau density when it is pivoted on. On average the most dense row or most dense column of a Hyper-sparse problem is less dense than the most dense row or most dense column of a Densefill problem. However there are both Hyper-sparse and Densefill problems that have very dense rows or columns. In these graphs (and all the graphs of Subnet

	singleton	singleton or doubleton
Hyper-sparse average	0.120	0.334
Densefill average	0.059	0.204

(a)



(b)

Figure 2.8: Subnet proportion of sparse columns in initial tableau. Averages in table (a) and values for each problem in graph (b)

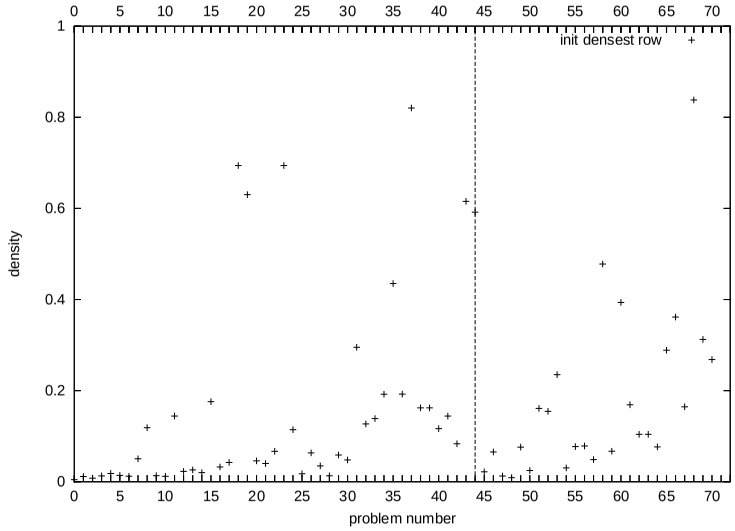
problems), the problems are ordered by initial density. These graphs therefore also show that the densest row is only loosely correlated with initial density (Figure 2.9(b)), whereas the densest column is more strongly correlated with initial density (Figure 2.9(c)).

We now consider the overall effect of the structure of the initial tableau. This is done by comparing the density of the optimal tableau from the original solve, the density of the optimal tableau from random structure solves, and the density of the optimal tableau estimated from a closed formula. The random structure solves and the closed formula are both new techniques.

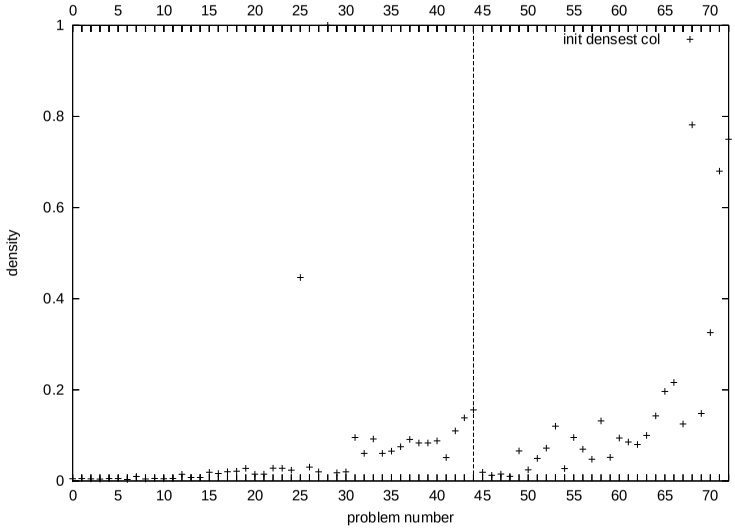
We first compare the original solve to the random structure solve. The random structure solve is similar to the random number solve above. We resolve each problem with an initial tableau that contains the same set of values as the original initial tableau but has them scattered randomly. The same number of pivots as in the original solve are performed on this tableau, randomly choosing a pivot element from among the nonzeros each iteration. This is repeated ten times with different initial tableau, often leading to very different final tableau densities. The average density of these optimal tableau for each problem, compared with the final density of the original solve, indicates whether or not the original sparsity pattern helps or hinders in

	row	column
Hyper-sparse average	0.163	0.069
Densefill average	0.236	0.162

(a)



(b)



(c)

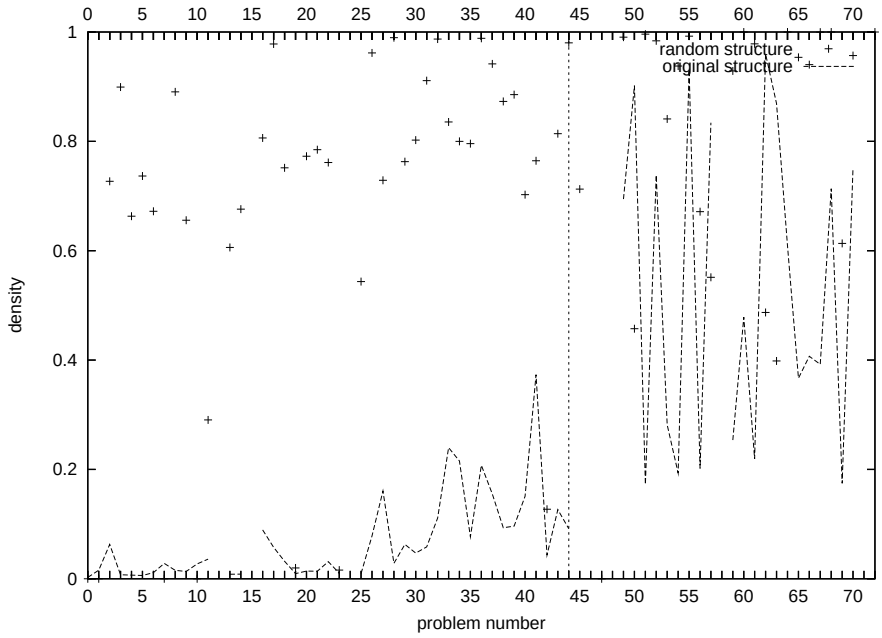
Figure 2.9: Subnet initial density of most dense row and column in initial tableau. Averages in table (a), value of most dense row in each problem in graph (b) and for most dense column in each problem in graph (c)

preserving sparsity.

Figure 2.10 shows the effect of structure on the Subnet problems. On average both Hyper-sparse and Densefill problems finished denser when solved with random structure. Solving

Times denser with random structure solve	
Hyper-sparse average	7.622
Densefill average	1.609

(a)



(b)

Figure 2.10: Subnet effect of structure. Averages in table (a) and results for each problem in graph (b)

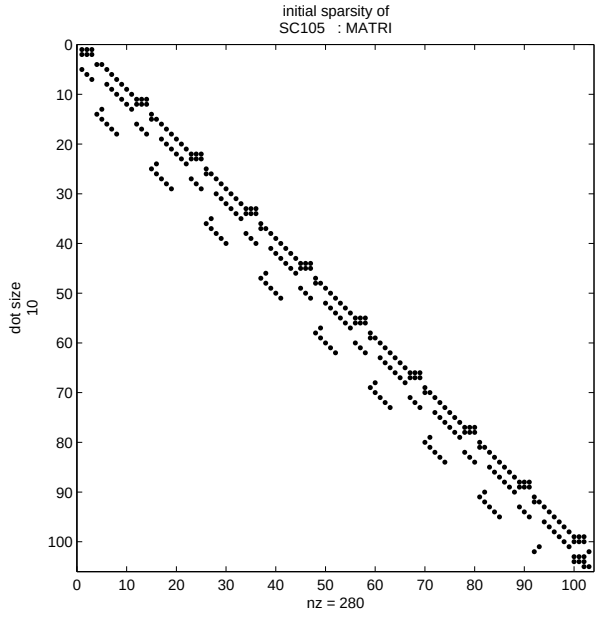


Figure 2.11: Initial tableau of SC105

with random structure had a much larger effect on Hyper-sparse problems, indicating that they benefit greatly from their structure. Nearly all problems finished denser on the random structure solves, with many close to fully dense. The only problems that finished less dense with the random structure solve were Densefill problems SC205 and SC105 (numbers 50 and 57), which have a staircase structure. Figure 2.11 shows the initial tableau of SC105. The number of nonzeros is shown below the matrix.

We now also consider the closed formula estimate. In the closed formula estimate of the density of the optimal tableau we make the assumption that at every iteration the tableau is uniformly randomly distributed with nonzeros, so every pivot row and pivot column is of density equal to the tableau. Although this is not expected to be true, the consequences of assuming it are illustrative. For ease of calculation we also assume that no numerical cancellation occurs. We begin with an all logical basis and perform a series of pivots that each add one structural to the basis, up to the correct number of structurals.

Recall that m and n are the number of rows and columns of the initial tableau respectively. Let C_1, C_2 and C_3 be constant functions of m and n (which are not defined here). Let a_k be the average number of nonzeros per column of the tableau, where k is the number of pivots so far, hence the number of structurals in the basis. a_0 is the average number of nonzeros per column of the initial tableau. k and a_k are implicitly linked by the formula below:

$$k \simeq C_1 \log\left(\frac{a_k - n}{a_0 - n}\right) + C_2 \log\left(\frac{a_k - 1}{a_0 - 1}\right) + C_3 \log\left(\frac{a_k n - m}{a_0 n - m}\right). \tag{2.4}$$

The derivation of the formula is in Section B of the appendix. Our formula can be used to plot the estimated density of a problem compared to the number of structurals in the basis. This density can be compared to the density when using the original structure, and to the random structure solve. Figure 2.12 shows these three densities, plotted against number of structurals in the basis, for Hyper-sparse problem AGG (number 38). The original solve is the least dense of the three, with the closed formula being by far the most dense. This result is typical for the Subnet problems. For all but five of the Subnet problems the closed formula suggests that the optimal tableau will be completely dense, often after very few pivots.

Each iteration of the original solves of Subnet problems typically has a Markowitz count of about what is expected, given the tableau density at that iteration. But the amount of fill in caused by each pivot is much lower than the Markowitz count suggests. This is partly due to numerical cancellation, but more importantly a significant amount of fill in does not occur as new nonzeros are created in the same position as old nonzeros. This is called *superposition*.

Superposition happens often with tableaux that come from real life problems. Suppose, for example, there are some constraints that involve nearly all variables, leading to a dense row in the initial tableau. A pivot on such a row makes copies of itself across the tableau. A later pivot

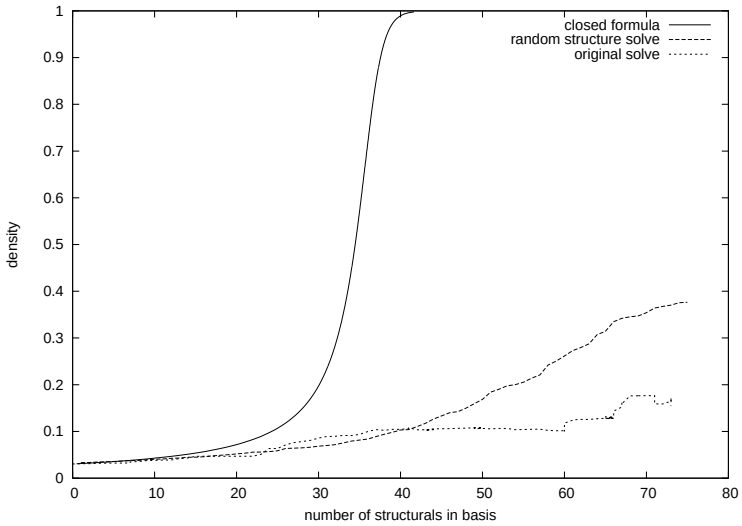


Figure 2.12: Density of AGG, compared to number of structurals in basis

somewhere in one of the copies will have a large amount of superposition with the other copies. The closed formula overestimates the density of the optimal tableau as it expects superposition to occur as if the nonzeros are uniformly randomly distributed in the tableau. In any actual simplex solve, including one with an initial random structure, there is more superposition than that. Even if the initial structure is random dense areas of the tableau will soon be formed. The fact that the Subnet problems have a more ordered structure than random explains why they often stay sparser than the random structure solves.

Among the Subnet problems it is the Hyper-sparse problems that have a structure that leads to the most superposition, hence the least fill in. For some Densefill problems the structure makes it immediately clear that there will be a lot of fill in. The staircase structure of problems like SC105 (Figure 2.11) means each pivot causes fill in outside the central band, leading to no superposition, and a rapid increase in density.

2.3 Basis structure

We mentioned above that the extended tableau at iteration k is given by $\boldsymbol{A'_k} = \boldsymbol{B_k}^{-1}\boldsymbol{N_k}$, where $\boldsymbol{B_k}$ and $\boldsymbol{N_k}$ are the basis and nonbasis at iteration k. Hence the basis matrix, and its inverse, directly affect the density of the tableau. In this section we look at what properties of the basis matrix cause its inverse, and consequently the tableau, to remain sparse or become dense.

To analyse the Subnet problems we permute basis matrices into what we call *Tarjan form*, introduced in Duff [1977]. This consists firstly of a symmetric permutation of the rows of the basis (or alternatively the columns) to get a matrix with entries on the main diagonal (the one from top left to bottom right). The existence of entries along the main diagonal is called a *transversal*. This is followed by an unsymmetric column and row permutation to give an *upper*

triangular block diagonal matrix. This is a matrix that has all entries permuted into the upper right half, as far as possible, and the remaining entries permuted to be within square blocks along the main diagonal. The Tarjan form is optimal, in that no diagonal blocks can be reduced into smaller diagonal blocks. We denote a basis $\boldsymbol{B}$ in Tarjan form as $\boldsymbol{B'}$.

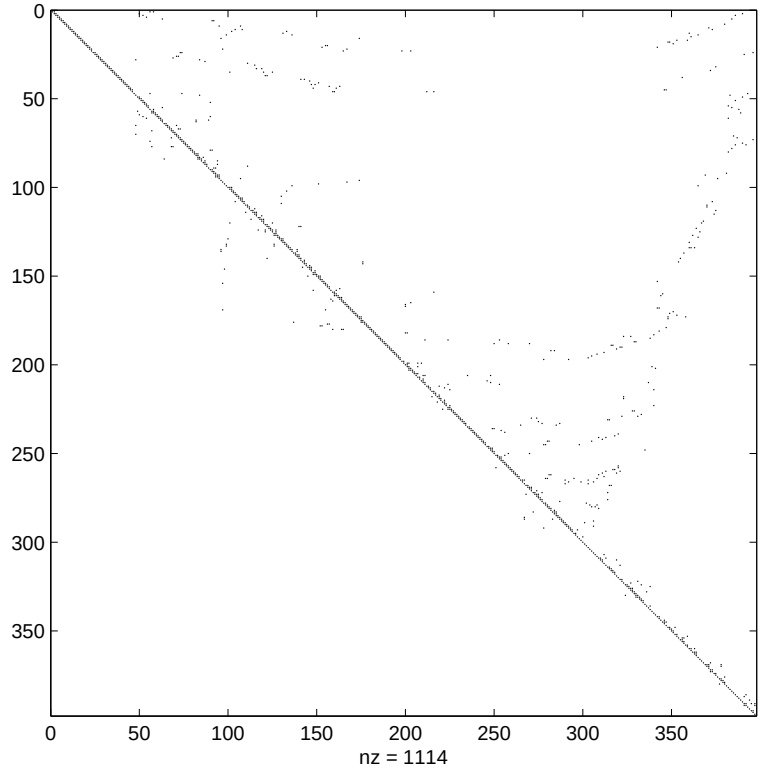


Figure 2.13: Optimal basis permuted into Tarjan form for SCSD8

Figure 2.13 shows a matrix in Tarjan form. Although most entries have been permuted to the upper right half of the matrix there are several *diagonal blocks*, of different sizes. These blocks are not solid, but consist of square areas along the main diagonal where nonzeros can occur. We call a diagonal block *trivial* if it is of size one by one, otherwise it is *nontrivial*. A matrix in Tarjan form is called *triangular* if it has no nontrivial diagonal blocks.

With a basis matrix in Tarjan form an instructive way to represent the matrix is as a graph, leading to the well known analysis below. Each row (or equivalently column) of the matrix can represent a node. These nodes are linked by directed arcs corresponding to the nonzero entries of the matrix, by forming an arc from node i to node j when element (i, j) of the basis is nonzero. Figure 2.14(a) and Figure 2.14(b) show an example matrix in Tarjan form and the representation of that matrix as a graph. As is customary when drawing the matrix as a graph we do not display all the trivial arcs from each node back to itself. If a single arc connects node i to node j we say there is a *direct path* from i to j, e.g. in Figure 2.14(b) there is a direct path from node 2 to node 5. If a series of directed arcs connects node i to node j we say there is an *indirect path* from i to j, e.g. in Figure 2.14(b) there is an indirect path from node 1 to node 4.

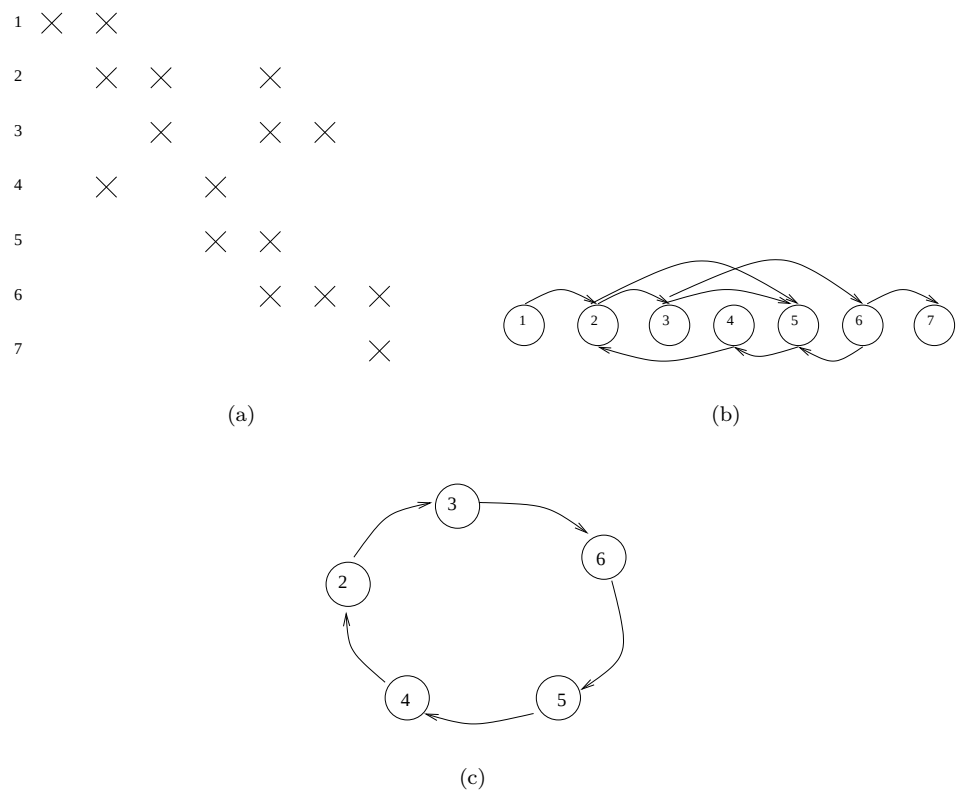


Figure 2.14: An example matrix in Tarjan form (a), shown as a graph (b), and one of the closed paths (c)

A nontrivial path from node i back to node i is called a *closed path*. Each closed path can be associated with a nontrivial diagonal block of a basis in Tarjan form. Collectively the nodes that make up a nontrivial diagonal block are known as a *strong component*. Figure 2.14(c) picks out the main strong component of the matrix. An arc from i to j with $i < j$ is called a *forward arc*, and corresponds to an entry above the diagonal in the upper right half of the basis. Similarly an arc from i to j with $i > j$ is called a *backward arc*, and corresponds to an entry below the diagonal in the lower left half of the basis. Each closed path must contain at least one backward arc and one forward arc. In a triangular basis all arcs are forward arcs.

We are interested in the density of the basis inverse. We refer to the inverse of a matrix, ignoring the effects of numerical cancellation, as the *symbolic inverse*. This is very easy to find with a matrix in Tarjan form. Although in an actual simplex solve the numerical cancellation does affect the sparsity (see Section 2.2.2), comparing densities of symbolic inverses still gives a good indication of which basis matrices will have sparse inverses.

The theorem below can be found in Duff et al. [1986] and elsewhere:

Theorem 1. *If a basis $\boldsymbol{B'}$ contains a direct or indirect path from i to j then the symbolic inverse $\boldsymbol{B'}^{-1}$ contains a nonzero at (i,j).*

A matrix in Tarjan form that has no large diagonal blocks has an inverse that is relatively

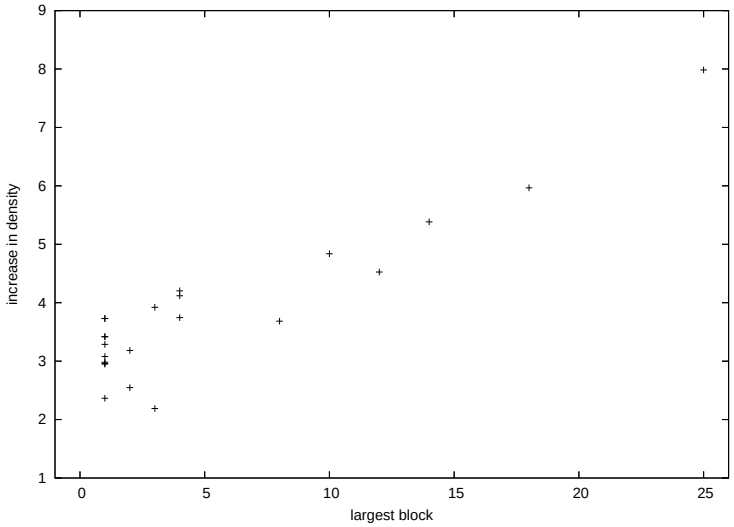
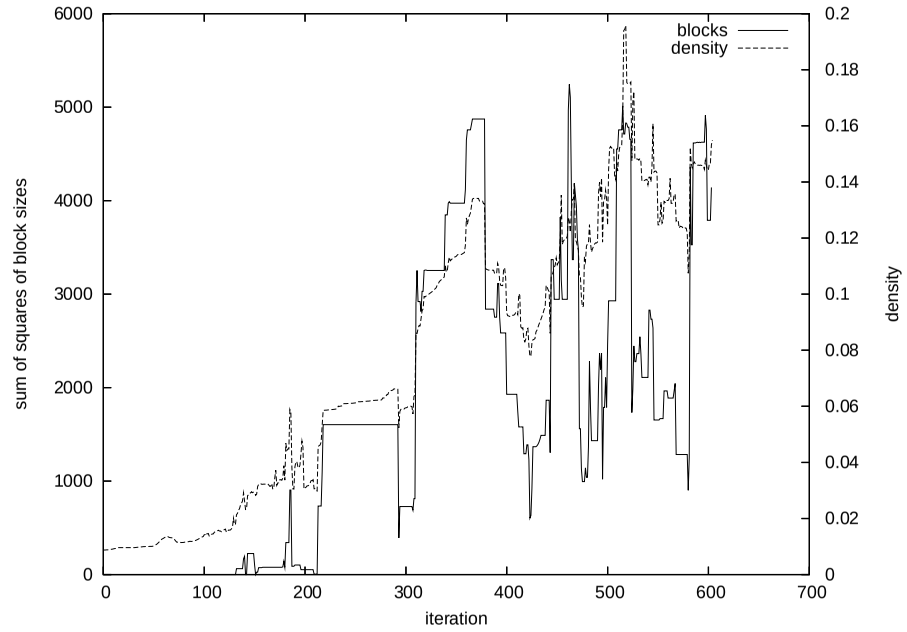


Figure 2.15: Increase in density when forming inverse

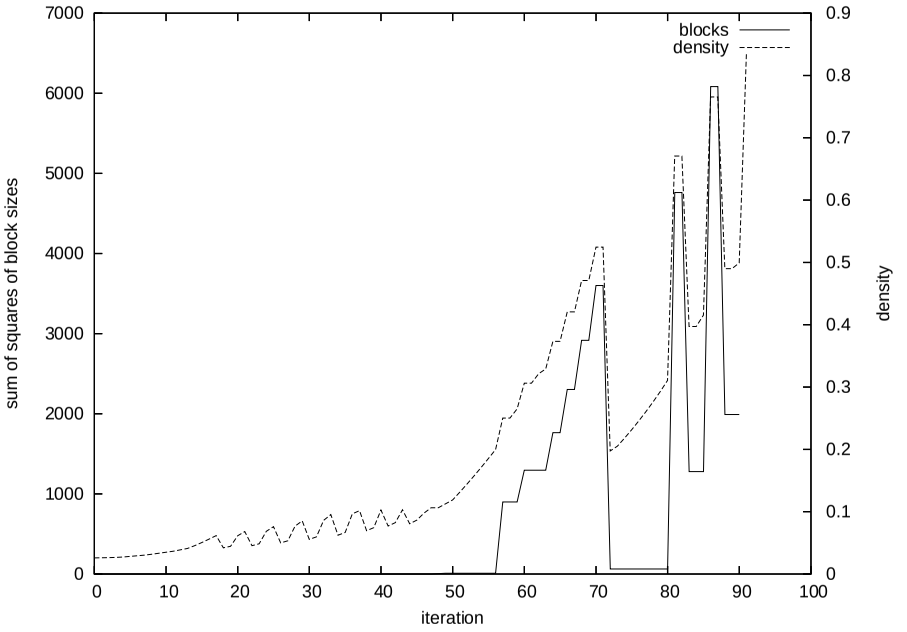
sparse. Figure 2.15 shows the increase in density when inverting each of a series of randomly generated matrices in Tarjan form. Matrices with a largest block of size one are triangular matrices. The graph shows that matrices with at least one nontrivial diagonal block have a more significant increase in density when inverted than those without. To further illustrate the correlation between block size and tableau density we look at two simplex solves. Figure 2.16(a) and Figure 2.16(b) show the sum of squares of nontrivial block sizes and the tableau density each iteration during the solves of Hyper-sparse problem ETAMACRO (number 27) and Densefill problem SC105 (number 57) respectively. The block sizes and tableau densities correlate closely for both problems.

Almost all problems, including Hyper-sparse ones, have a nontriangular optimal basis. However, the optimal basis of most Hyper-sparse problems rarely has large diagonal blocks, and instead often has many small diagonal blocks. Figure 2.17 shows the optimal basis of the Hyper-sparse problem GANGES (number 8) as an example of this. Densefill problems often have large diagonal blocks in the optimal basis, often of size about 50% of the basis. Problems in the SC family, such as SC105, shown in Figure 2.11, quickly become dense. After a few simplex pivots of SC105 over 90% of the columns of the basis are part of one very large diagonal block.

We now give an instructive way to view the nonbasis. This rearrangement of the nonbasis is similar to the rearrangement that gave the Tarjan form for the basis. Our nonbasis form is believed to be original, though a similar form is seen in Saunders [1976]. We first apply to the nonbasis the same unsymmetric row permutation that was applied to the basis when getting the Tarjan form. We then permute the columns, ordering them by the entry that has the lowest placement in the column. We call the resulting form the *permuted nonbasis*, denoted $\boldsymbol{N'}$. Figure 2.18 shows the basis in Tarjan form and permuted nonbasis, at the optimal solution to Hyper-sparse problem LOTFI (number 35). The basis is triangular, though this is not the



(a)



(b)

Figure 2.16: Tableau density and block sizes in Tarjan form each iteration when solving ETA-MACRO (a) and SC105 (b)

case throughout the solve of this problem. We refer to the area of the permuted nonbasis that contains the nonzeros the *sloped section*, and refer to leading edge of the sloped section as the *profile*. When there is an entirely structural basis the sloped section contains an entire (permuted) identity matrix. Even if this is not the case there is always at least one nonzero per

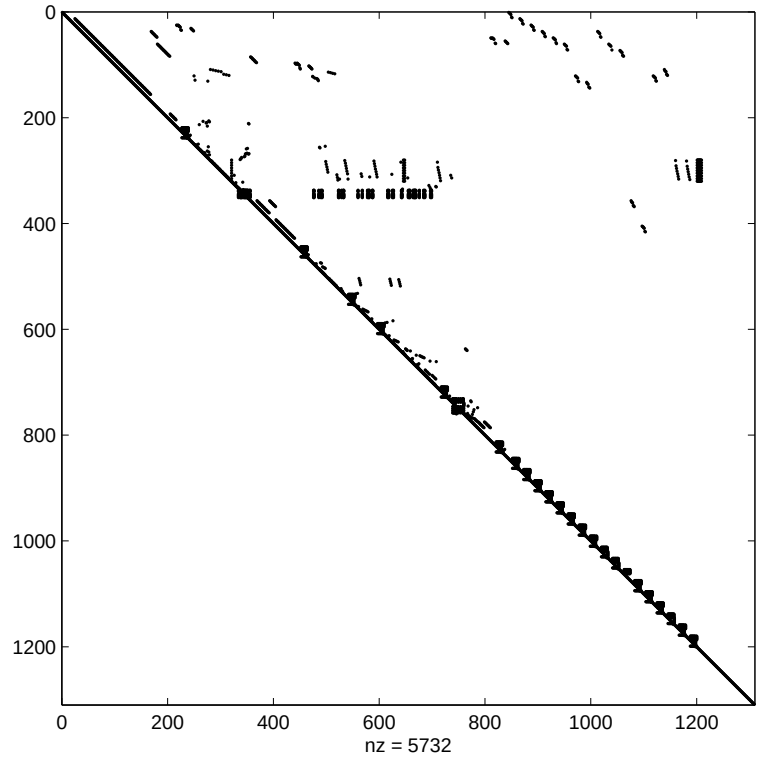


Figure 2.17: Optimal basis permuted into Tarjan form for GANGES

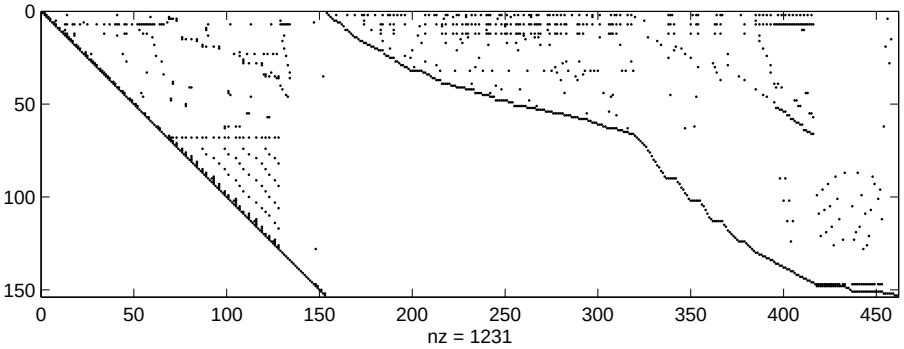


Figure 2.18: Optimal basis and permuted nonbasis for LOTFI. This is a rearrangement of the optimal extended tableau to give $(\boldsymbol{B'} : \boldsymbol{N'})$

row of the sloped section, hence the profile is smooth.

To form the tableau the basis is inverted and multiplied by the permuted nonbasis, hence the profile of the nonbasis is important in indicating the expected density of the tableau. In particular when the basis is triangular, as in Figure 2.18, fill in can only occur in the sloped section of the nonbasis. An example of this can be seen by comparing Figure 2.18 and Figure 2.19.

We saw above how the Subnet problems, and especially Hyper-sparse problems, preserve sparsity better than random structure problems. It is also true that they have a more gently sloped profile, and hence a smaller sloped section. In a real problem, with nonrandom structure,

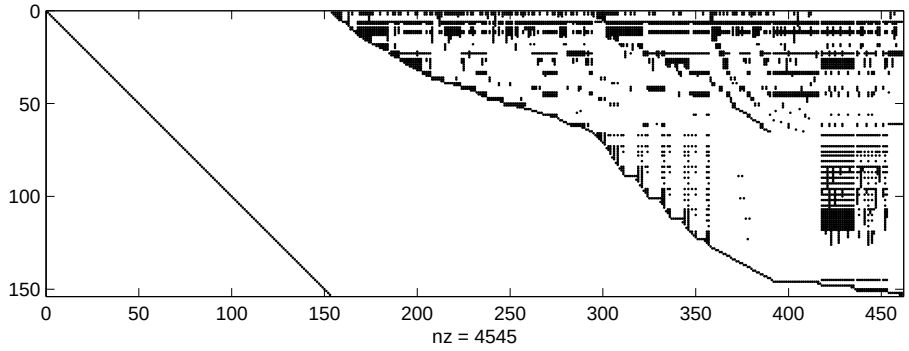


Figure 2.19: Identity matrix and optimal tableau of LOTFI, formed by premultiplying $(\boldsymbol{B'} : \boldsymbol{N'})$ of Figure 2.18 by $\boldsymbol{B'}^{-1}$, to give $(\boldsymbol{I} : \boldsymbol{B'}^{-1}\boldsymbol{N'})$

there is a variation in row density. When forming the Tarjan form the dense rows tend to get placed towards the top of the basis, as the Tarjan form attempts to form an upper triangular basis. These dense rows are therefore also placed towards the top of the nonbasis, leading to a gentle profile.

2.3.1 Inverting the basis

We now look in more detail at how the structure of the basis affects the density of the basis inverse.

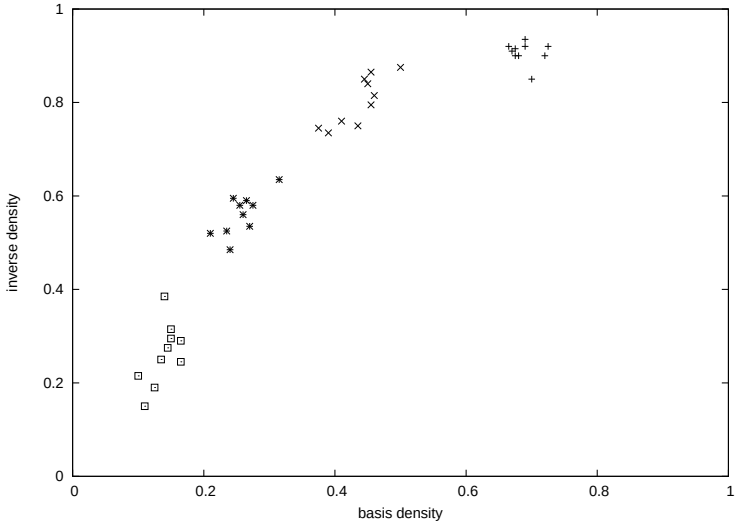


Figure 2.20: Inverting triangular matrices

Figure 2.20 compares the basis density and the inverse density for several randomly generated triangular matrices. The graph shows that the relative initial density of a basis matrix is a good indicator of the relative density of the inverse, as denser basis matrices tend to have denser inverses. However the correspondence is not exact, as sometimes less dense basis matrices lead to more dense inverses. The basis matrices that lead to the largest increases in inverse density were those with an especially dense area, and those with many entries close to the diagonal.

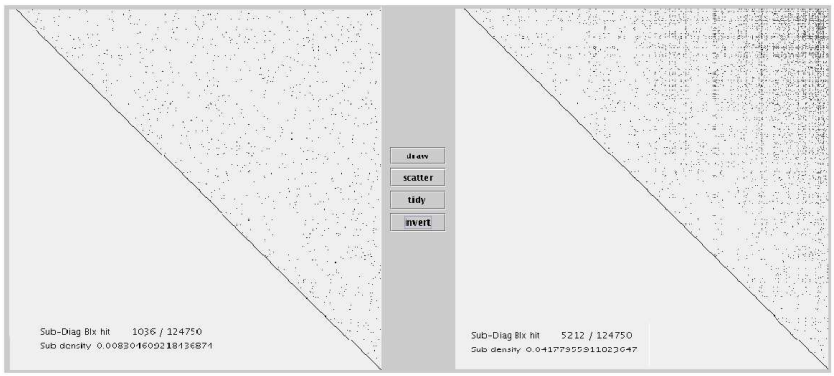
This leads to the following observation.

Observation 1. *In a triangular matrix entries close to the diagonal cause a high level of fill in in the inverse.*

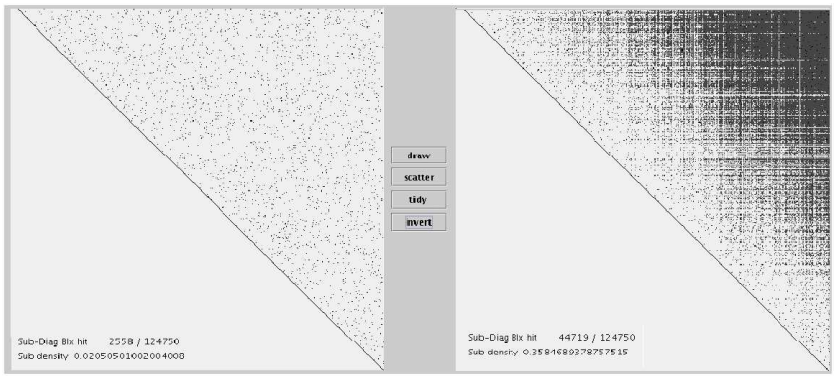
When the basis is viewed as a graph entries close to the diagonal correspond to direct paths linking nodes that are adjacent (e.g. nodes 7 and 8) or close to adjacent (e.g. nodes 13 and 16), which can therefore appear on a great many closed paths. Figure 2.21 demonstrates where fill in is most likely to occur when inverting triangular basis matrices. Each of the four graphs shows a random upper triangular matrix and its inverse. In all cases the inverses have the most fill in far from the diagonal. This leads to the observation below, the counterpart of Observation 1:

Observation 2. *In a triangular matrix elements far from the diagonal are likely to be filled in in the inverse.*

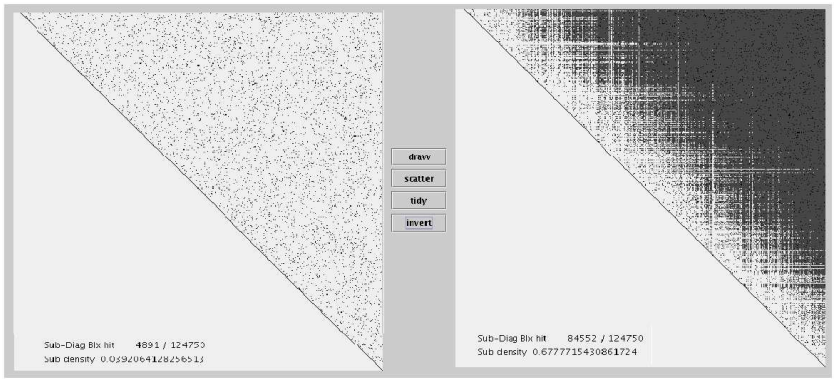
Indeed the probability of an element in the inverse being filled in depends only on its distance from the diagonal. The criss-cross pattern in the inverses of Figure 2.21 occurs as certain sparse rows (and columns) in the basis matrices remain sparse in the inverse. These rows (or columns) can be thought of as nodes that are only loosely connected, or even unconnected, to the rest of the graph.



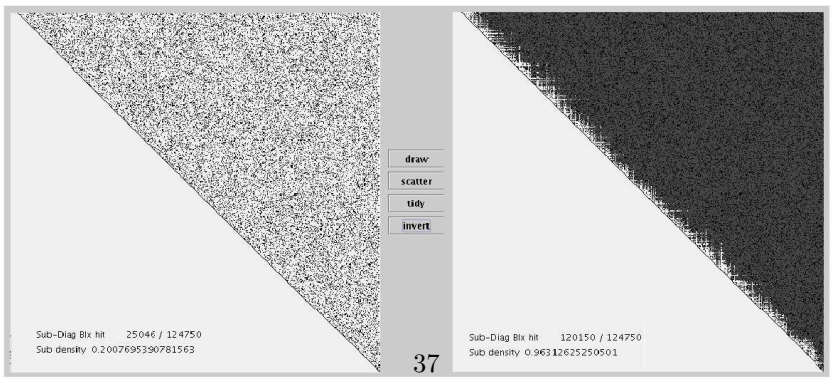
(a)



(b)



(c)

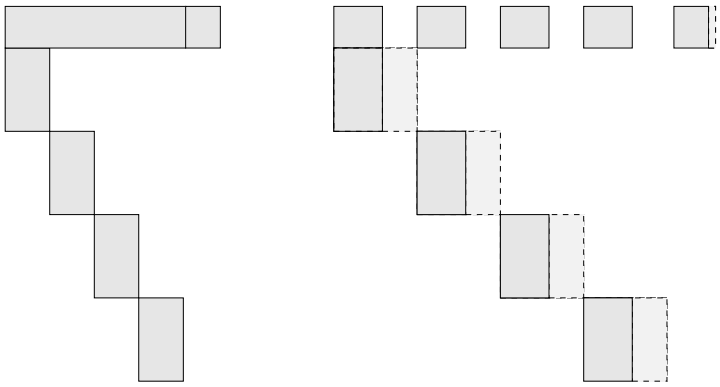


(d)

Figure 2.21: Random upper triangular basis matrices of differing density and their inverses. The basis matrices have 2 (a), 5 (b), 10 (c) and 50 (d) nonzeros per column

2.3.2 Block structured problems

We now look at how the block structure of an initial tableau is to some extent preserved in the basis matrix. We will show that the block structure of a problem has implications for the maximum density of the tableau.



(a) (b)

Figure 2.22: Block structured initial tableau (a) and corresponding basis matrix (b)

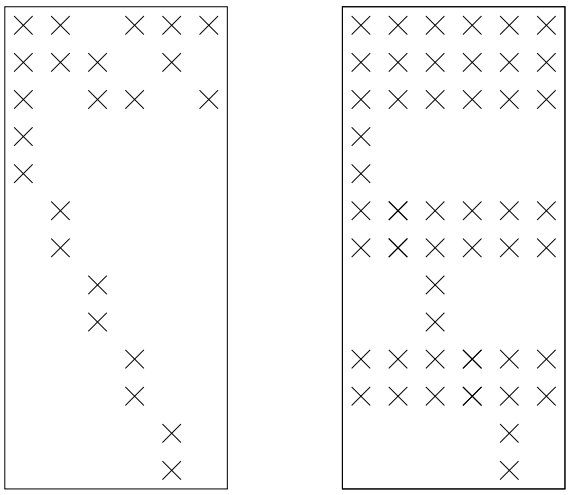
Figure 2.22(a) shows an initial tableau with block structure, specifically *row linked* block structure. All of the nonzeros appear in the diagonal blocks, apart from those in the rows across the top of the problem. These rows are called *linking rows*, as they link together the otherwise separable diagonal blocks. At the right hand edge of the linking rows is what is called the *private block*. This is a section of the linking rows that has no diagonal blocks beneath it. Let n be the total number of rows, l be the number of linking rows and b the number of diagonal blocks. For simplicity we assume that $l \leq b$, and that all diagonal blocks have the same number of rows, specifically they all have $\frac{n-l}{b}$. Since the basis matrix always consists of columns from the initial tableau, and from the identity matrix, it may also be represented as having a block structure. Figure 2.22(b) shows the basis, drawn to emphasise the similarity in structure with the initial tableau. The darker areas in Figure 2.22(b) are either columns from the initial tableau or identity columns, and the lighter areas are all identity columns.

At any time in a simplex solve each column of the basis corresponds to one row of the extended tableau. We say that each row is *basic in a column*. Our implementation of the tableau simplex method begins with an all logical basis, so every row is basic in the column of a logical variable. During a solve a row can be pivoted on several times, changing the column in which that row is basic.

For a row linked problem there are only three possibilities for where each linking row is basic in the optimal solution. The first possibility is that the row is still basic in the logical column. The second possibility is that the row is basic in a column of the private block. The third possibility is that the row is basic in a column of a diagonal block. This third possibility

only happens if there has been a pivot in the linking row above a diagonal block. These pivots can have the effect of copying the diagonal block across the tableau, causing a large increase in tableau density. At any time though the maximum number of linking rows that are basic in the column of a diagonal block is the number of linking rows. This leads to the original theorem below:

Theorem 2. *The maximum tableau density outside the linking rows is approximately* $\frac{l}{b}$.



(a) (b)

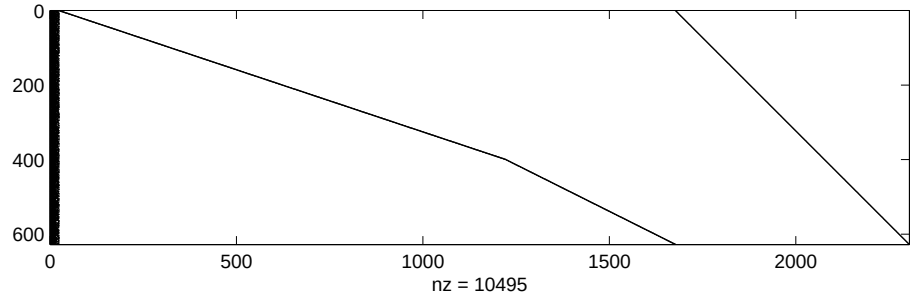
Figure 2.23: Row linked block structured problem initial tableau (a) and example optimal tableau (b)

We give a demonstration before a proof of the theorem. Figure 2.23(a) shows the initial tableau of a row linked block structured problem. There are $n = 13$ rows, $l = 3$ linking rows, $b = 5$ diagonal blocks and a small private block. Figure 2.23(b) shows a possible optimal tableau. The linking rows have all become dense, and there have been pivots in two of them above diagonal blocks, causing the diagonal blocks to be copied across the tableau. The density of the optimal tableau outside the linking rows is about $\frac{2}{5}$, less than the potential maximum of about $\frac{3}{5}$.

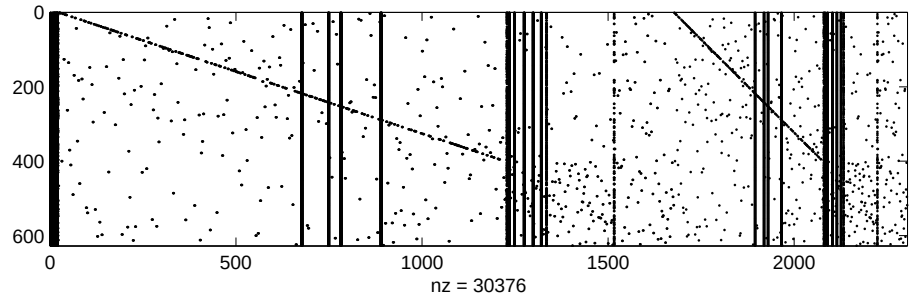
In order to prove Theorem 2 we first note that the optimal tableau is the result of the partition of variables into basic and nonbasic, so any order of pivots that results in the same partition as the optimal tableau is equivalent. We now consider constructing the optimal partition, beginning with an all logical basis, and following a special pivot order. The special pivot order is to perform all the *diagonal block pivots*, those within the diagonal blocks, before all the *linking row pivots*, those within the linking rows but not in the private block.

The diagonal block pivots only cause fill in within the diagonal blocks, and within the linking rows. The linking row pivots cause significant fill in, with each one causing at most one dense band of $\frac{n-l}{b}$ rows across the matrix. As there are only l linking rows there can be only l dense

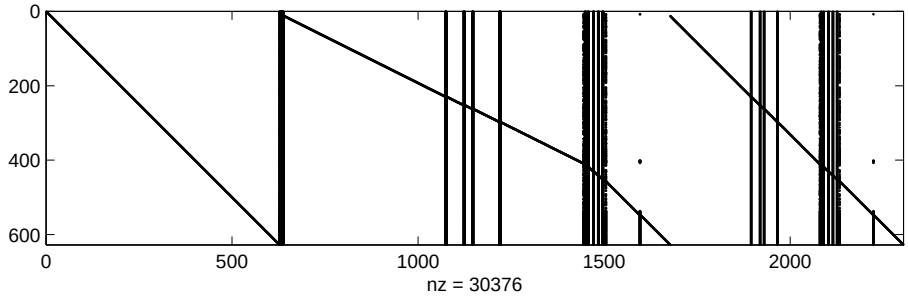
bands created, leading to $\frac{(n-l)l}{b}$ nonzeros in the dense bands. Discounting the nonzeros in the diagonal blocks themselves this gives a density in the $n-l$ rows outside the linking blocks of $\frac{l}{b}$, proving the theorem. Pivots in the private block, whenever they occur, do not create fill in outside the blocks and linking rows. Given specific block dimensions a more exact bound than $\frac{l}{b}$ can be found.



(a)



(b)



(c)

Figure 2.24: FIT1P initial extended tableau (a), optimal extended tableau (b), and optimal extended tableau reordered (c)

Theorem 2 can also be applied to column linked problems, with l this time being the number of linking columns. Figure 2.24(a) shows the initial extended tableau of the column linked Hyper-sparse problem FIT1P (number 28) (which includes an identity matrix as this is

the extended tableau). Apart from the first few columns of FIT1P, which are almost completely dense, the remaining columns each have only one nonzero in them. There is no private block. Figure 2.24(b) is the optimal extended tableau. This is the tableau produced at the end of the simplex solve. It does not have a clear pattern of nonzeros. Figure 2.24(c) is a column reordering of the optimal extended tableau. The first 627 columns of Figure 2.24(c), which includes the first band of dense columns, are the basis. This is followed by the nonbasis, which is ordered to first show the 1050 nonbasic structural variables then the 627 nonbasic logical variables. The reordered optimal extended tableau shows that there are just a few bands of dense columns, where the linking columns have been pivoted on, and the rest of the matrix is still identity columns.

2.4 Summary

In this chapter we have created the set of Subnet problems, which is divided into Hyper-sparse and Densefill problems. The Subnet problems were analysed to determine why some problems stay sparse when solved with the simplex method, and can therefore be solved with comparatively little work. Hyper-sparse problems were found to typically solve in fewer iterations with a greater proportion of bound swap iterations. They finished with fewer structurals in the basis, and benefited more greatly from numerical cancellation.

Across both problem sets the density of the optimal tableau was markedly less than the density predicted by solving with an initial tableau with random structure, or the density predicted from using a formula that assumed random uniform distribution of nonzeros in the tableau each iteration. This indicates that the position of the nonzeros in the initial tableau is significant in preserving sparsity. Indeed the structure of the tableau was found to be the most important indicator of whether or not sparsity was preserved. This underlines the importance of the structure of a problem in choosing a solution approach.

The density of a tableau is linked to the density of the inverse of the basis matrix. When the basis matrix is viewed in Tarjan form the number and size of diagonal blocks affects the inverse density. Hyper-sparse basis matrices in Tarjan form were found to occasionally be triangular and often have only a few small diagonal blocks. Densefill basis matrices often in Tarjan form often had large diagonal blocks. A theorem was given to explain the limited increase in density of block structured row and column linked problems.

Chapter 3

Row and Column Linked LPs

In this chapter we look at using the tableau simplex method to solve row linked or column linked LPs, while preserving the sparsity of the tableau. This follows on from Chapter 2, and uses insights into why some problems become dense to motivate the algorithms. This is of direct benefit to solving with the tableau simplex method, but more importantly offers insight into reducing the work required when solving with the revised simplex method.

There are three new algorithms for solving columns linked problems, and four new algorithms for solving row linked problems. We then combine the best new row and column algorithms to effectively solve the whole Subnet set of problems, in Solve 5 below. This is all with the tableau simplex method. In Solve 6 the algorithm of Solve 5 is adapted for the revised simplex method, and in Solve 7 a further revised simplex method variant is proposed.

3.1 Introduction

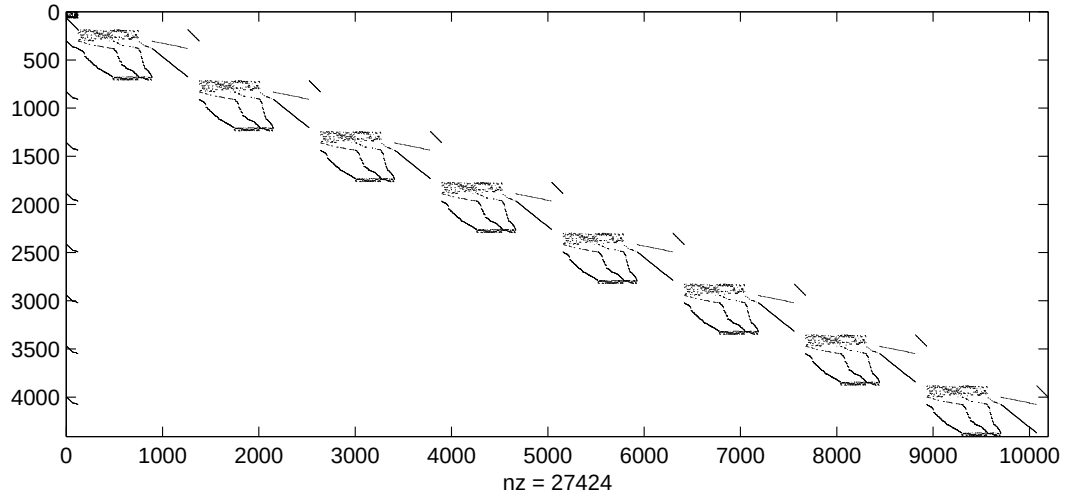


Figure 3.1: STORMG2-8 initial tableau. Columns 1 to 121 are linking

Many real life problems, including several in Subnet, are block structured with linking rows or linking columns. We divide the linking rows or columns into two types. Sometimes, as in FIT1P of Figure 2.24(a), the linking rows or columns are relatively dense, and so pivoting on them directly causes a large amount of fill in. In this case we refer to them as *dense linking rows* or *dense linking columns*. If, as in STORMG2-8 of Figure 3.1, the linking rows or linking columns are only undesirable to use as pivots as they link together otherwise separable parts of the tableau, we simply refer to the them as linking rows or linking columns.

It should be noted that although we classify some problems as row or column linked, every LP can be *dualised* resulting in a *dual problem*, which has the same optimal objective value but a different structure. Dualising a row linked problem transforms it into a column linked problem, and vice versa. For simplicity we only consider problems with their original structure, and do not dualise them. Furthermore some problems without an immediately apparent row or column linked structure may be rearranged to have one, see e.g. Ferris and Horn [1998]. This means that the methods introduced below may be applicable to more problems than those that are not immediately obvious as row or column linked.

In Section 3.2 we introduce methods for preserving tableau sparsity while solving column linked problems. In Section 3.3 we introduce methods for preserving tableau sparsity while solving row linked problems. In Section 3.4 we solve all the Subnet problems using some of these methods. All LPs are solved with variations of the tableau simplex solver Simplex10.

Most of the solve methods described below are designed specifically for row or column linked problems. They require the location of the linking rows or columns in the initial tableau. Although this detection can be done by the user, this can also be done automatically to some extent, see e.g. Robert E. Bixby [1988]. These solve methods also do not require explicit knowledge of the tableau each iteration (only at the first iteration), so can be readily adapted for the revised simplex method.

A few solve methods, specifically the methods of *avoiding dense columns* and *avoiding dense rows*, do not require user input on the structure of the initial tableau. These two solve methods can therefore be applied to all the Subnet problems, which we do in Sections 3.4.1 - 3.4.3. However, these two solve methods do require explicit knowledge of the tableau each iteration, so cannot be readily adapted for the revised simplex method. In Section 3.4.4 we show how these solve methods could be adapted for use with the revised simplex method. In Section 3.4.5 we consider another new solve method suitable for the revised simplex method.

3.2 Column Linked Problems

In this section we look at solving column linked problems, in particular FIT1P and STORMG2-8. FIT1P (shown in Figure 2.24(a)) has 24 dense linking columns and identity columns elsewhere. STORMG2-8 (shown in Figure 3.1) has eight similar diagonal blocks loosely joined by

121 linking columns. The difference between the problems in the nature of their linking columns affects which solution methods are effective on each problem. A general column linked LP, with k blocks of variables, may be written in the form:

$$\begin{aligned} &\min \sum_{i=1}^{k} \boldsymbol{c_i}^T \boldsymbol{x_i}, && (3.1)\\ &\text{s.t. } \boldsymbol{A_{i1}x_1} + \boldsymbol{A_{ii}x_i} \leq \boldsymbol{b_i} && i \in 1 \ldots k,\\ &\quad \boldsymbol{x} \in \boldsymbol{X}, \end{aligned}$$

where $\boldsymbol{x_1}$ are the linking columns, and $\boldsymbol{A_{ij}}$ is a sub-block of the constraint matrix $\boldsymbol{A}$. One method to solve column linked problems is Benders decomposition (Benders [1962]), for which an outline is given below.

1 Fix the value of the linking columns.
2 Solve the resulting series of LPs.
3 If all LPs are feasible add an *optimality cut* to the master problem, else add a *feasibility cut*.
4 Solve the master problem, to find new values for the linking columns.

The linking columns are relaxed, leaving a series of small LPs. These small LPs are repeatedly solved, with each solve giving a linear constraint that is added to a *master problem*. The master problem progressively finds the optimal value of the linking columns.

In this section we introduce three new methods for columns linked problems, that do not relax the linking columns. Instead we leave them in the problem but steer the simplex method away from pivoting on them. In the simplex method there is typically a wide choice of acceptable pivot columns (see e.g. Terlaky and Zhang [1993]), and to a lesser extent a choice of acceptable pivot rows. Most of the various rules given in the literature for selecting pivot columns are motivated by trying to reduce the number of iterations. We instead try to select pivot columns that preserve sparsity. Our work is similar to Lentini et al. [1995], where *to preserve the sparsity ... a careful but heuristic choice* of the pivot row and column is made.

Our three methods are each tested on only two column linked problems, but the most promising of the methods is then tested on all Subnet problems in Section 3.4.

3.2.1 Avoiding linking columns

This method alters the pivot column selection rule to only pivot on linking columns when no other columns are attractive.

This is similar in spirit to the basis suppression method of Patty [1989], where in a problem with a single linking column that column is prevented from entering the basis. In the method of avoiding linking columns a set of vetoed columns is created, and initialised with the linking

columns. Depending on the type of problem solved this set may be kept constant or could be updated each iteration. When choosing the pivot column the most attractive column not in the vetoed set is chosen. If the only attractive columns are within the vetoed set the most attractive of these is chosen.

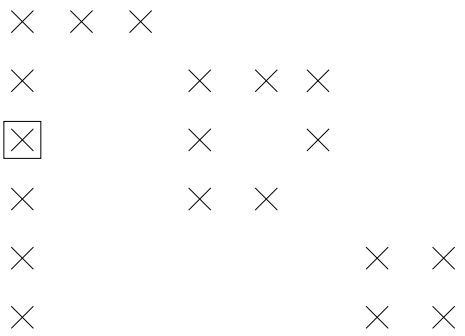


Figure 3.2: Tableau with dense linking column

For problems with dense linking columns the vetoed set is updated during the solve, by adding columns that become dense after a pivot from within the vetoed set. We call this *avoiding and updating*. Figure 3.2 shows an example tableau with a pivot element on a dense linking column, indicated by the square. The initial vetoed set is $\{1\}$. If we are avoiding and updating after pivoting where indicated the vetoed set is updated to $\{4, 6\}$, since these columns become dense after the pivot.

	FIT1P		STORMG2-8	
	pivots	work	pivots	work
standard solve	1350	14.0M	4979	11.2M
avoiding linking columns	1375	13.0M	5140	8.6M
avoiding and updating	989	10.2M	5963	25.4M

Table 3.1: Results for avoiding linking columns

Table 3.1 shows results for solving FIT1P and STORMG2-8 by three different methods. We compare the standard tableau simplex method solve, the solve avoiding linking columns, and the solve avoiding and updating. For each method for each problem we give the amount of work, defined as the total of the products of nonzeros in the pivot column and the pivot row each iteration, and the number of pivots. For FIT1P, the solve avoiding linking columns takes more pivots but less work than the standard solve. The standard solve pivots on a linking column on the very first iteration. The solve avoiding linking columns postpones any pivots on linking columns until the 628th iteration, then continues by optimising the rest of the problem, pivoting on another vetoed column, optimising the rest of the problem, and so on until optimality. In total 15 of the 24 vetoed columns are pivoted on. The solve avoiding and updating decreases the number of pivots and further decreases the work for this problem. For STORMG2-8, the solve avoiding linking columns also takes more pivots but less work than the standard solve. However, as this problem has linking columns which are not dense, the solve avoiding and

updating is not appropriate, and actually increases the work significantly. This is because it vetoes a very large proportion of the columns, leading to a long series of dense pivots at the end of the solve. Although it is usually good to postpone dense pivots until the end of the solve in this case there were so many that the amount of work increased.

Although it showed promising results on the two test problems the method of avoiding linking columns is only applicable to column linked problems, so is not tested on the whole Subnet set of problems.

3.2.2 Avoiding dense columns

This method alters the pivot columns selection rule to be biased against dense columns.

As stated above the default pivot column selection rule in each iteration of Simplex10 is to choose the column i with the largest score s_i, where the scores are calculated by steepest edge pricing. With the method of avoiding dense columns we alter the scores by taking into account the number of nonzeros in column i, referred to as the *column nonzeros* and denoted by z_i^{col}. The effect of the column nonzeros is weighted by a parameter ν^{col}. The new column score for column i is:

$$s_i' = s_i (z_i^{col})^{-\nu^{col}}, \tag{3.2}$$

where $\boldsymbol{z^{col}}$ is calculated from the initial tableau then updated each iteration. Updating $\boldsymbol{z^{col}}$ can be done cheaply when using the tableau simplex method.

	FIT1P		STORMG2-8	
	pivots	work	pivots	work
standard solve ($\nu^{col} = 0$)	1350	14.0M	4979	11.2M
$\nu^{col} = 1$	1162	11.1M	5728	9.2M
$\nu^{col} = 2$	1257	9.4M	6149	9.8M
$\nu^{col} = 3$	1224	8.3M	6306	11.2M

Table 3.2: Results for avoiding dense columns

Table 3.2 shows results for solving the two test problems. We compare the standard solve with three solves varying the size of the parameter ν^{col}. For FIT1P, the number of pivots and work decreased for all nonzero values of ν^{col} tested. The largest value of ν^{col} reduced the work most significantly. STORMG2-8 solved well for smaller values of ν^{col}, increasing the number of pivots but reducing the work. However for $\nu^{col} = 3$ STORMG2-8 did not solve as well, as at the end of the solve there were very many dense pivots.

The method of avoiding dense columns is effective on both these test problems, and we will see later (in Solve 4 of Section 3.4 and onwards), that this simple method is generally effective across the whole Subnet set.

3.2.3 Pushing columns

This method alters the way some pivots are performed so that pivots on linking columns do not affect the tableau density. It can be viewed as an extension of avoiding the linking columns.

In this method a set of vetoed columns is created, and initialised with the linking columns. If the linking columns are dense this set is updated each iteration, if they are merely linking the set is not updated. When choosing the pivot column if there is no attractive unvetoed column we choose the most attractive vetoed column. When this happens rather than doing a normal pivot we instead *push* the column. Pushing a column means adjusting the value of that variable by the amount suggested by the simplex method, but keeping the column nonbasic by not performing the pivot. After we push a vetoed column we continue by pushing other vetoed columns, until either an unvetoed column is made attractive, in which case the simplex method resumes, or, if no progress can be made by attempting to push each vetoed column in turn, a pivot is made on the most attractive vetoed column. We set a limit on the total number of times an attempt can be made to push each column. Without this limit a very long series of small pushes can occur towards the end of a solve which is often more time consuming than just performing a pivot. In the tests below we set the limit to be 1000 attempted pushes per column.

The method of pushing columns is more effective if we are able to push the columns further. The default row selection rule in Simplex10 is to only allow pivots that increase neither the sum nor the number of infeasibilities. To enable larger pushes we adjust the pivot row selection rule so that it is possible to select a row that results in an increase in the number of infeasibilities (as long as the sum of infeasibilities still decreases), and even allow pivot rows that increase the sum of infeasibilities, if they give a good enough improvement in the objective value. To prevent this rule causing cycling, the column selection rule is also slightly adjusted.

Pushing a column is like performing a bound swap, but involves moving a variable to a bound that did not previously exist. Ideally after pushing columns a new non-vetoed nonbasic column is made attractive, and the simplex method can continue without ever needing to pivot on a vetoed column. But progress can also be made even if no new nonbasic columns become attractive, as a series of pushes can move the solution towards optimality. Trying to find the optimal values of vetoed columns, by pushing them and not performing a pivot, is a *coordinate axes search*, meaning the variables are partitioned into sets and one set is optimised at a time while the others are held fixed. Here we change the value of only one vetoed column at a time. The effectiveness of the method therefore depends on how much the vetoed columns are able to move relative to each other, and the method will not be effective if the vetoed columns are closely linked and cannot move far independently.

Figure 3.3 shows an example LP where pushing would fail. The current vertex is not optimal, but to get the optimal vertex both x_1 and x_2 must be moved simultaneously. Decreasing x_1

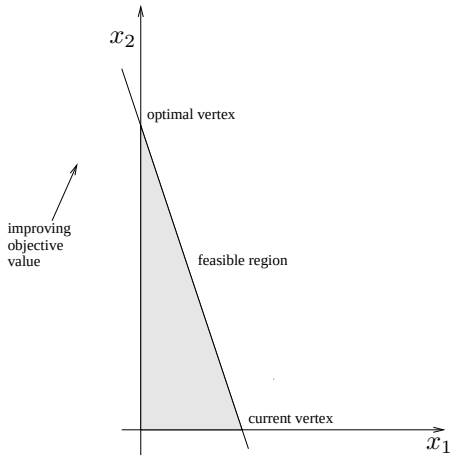


Figure 3.3: Example feasible region

independently leads to a worse objective value, and increasing x_2 independently makes the problem infeasible.

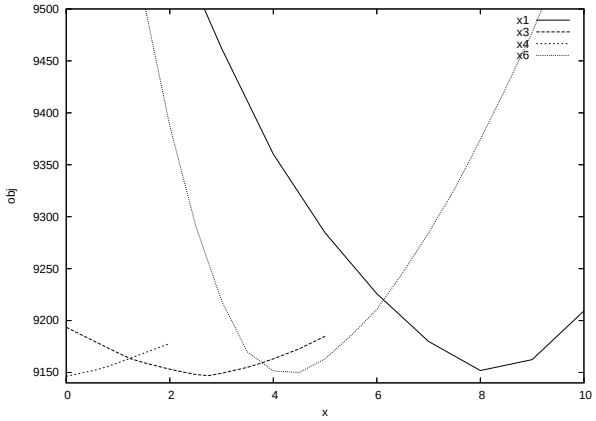


Figure 3.4: Four response curves for FIT1P around the solution

Figure 3.4 shows the *response curve* for four of the linking columns of FIT1P close to the optimal solution ($f^* = 9146$). The response curve for each variable is the best objective value for the problem as a function of that variable's fixed value. In an LP all response curves are *piecewise linear* and *convex*. A piecewise linear curve is one made up of sections, of which each section is linear (a straight line). A function is said to be convex if it has nonnegative curvature at all points, such as $f(x) = x^2$. As their curvature is zero at all points, all linear functions are convex. We call the points on the response curves where there is a sudden change in gradient *break points*. The break points in the response curves correspond to situations, such as the one shown in Figure 3.3, where a coordinate axes search can stall. However, the response curves displayed in Figure 3.4 seem reasonably smooth, with only a small change in gradient between each section, suggesting that the method of pushing may be successful on FIT1P.

Table 3.3 compares the standard solve (with the adjustments to the pivot row selection rule suggested above), with the solve pushing columns (also with these adjustments of course).

	FIT1P		STORMG2-8	
	pivots	work	pivots	work
standard solve	989	10.2M	5085	8.5M
pushing columns	1074	8.1M	5077	8.2M

Table 3.3: Results for pushing vetoed columns

For FIT1P, which has dense linking columns, the set of vetoed columns was updated each iteration. In this problem some pushing was possible, and sometimes progress was made from several vetoed columns being pushed in turn. The pushes were often small though, particularly towards the end of the solve. The optimal solution to FIT1P is *degenerate*, meaning there are many possible optimal solutions. Because of this the solve pushing columns was also able to finish with a less dense optimal tableau, as it pivoted on fewer vetoed columns. For STORMG2-8, which does not have dense linking columns, the vetoed set was not updated each iteration. On this problem very little pushing was possible, except for a small amount during phase two, which had little impact on the solve. Thus pushing is effective on only one of our two test problems.

Although pushing was shown to be effective on F1T1P, the method is only applicable to column linked problems, so is not tested on the whole Subnet set of problems.

3.3 Row linked problems

In this section we look at solving row linked problems. We look in particular at the three problems shown in Figure 3.5. Although these problems are all row linked, the row linked structure of KEN-07 is not obvious without a rearrangement of the tableau. A general row linked LP, with k blocks of variables, may be written in the form:

$$\begin{aligned} &\min \sum_{i=1}^{k} \boldsymbol{c_i}^T \boldsymbol{x_i}, && (3.3)\\ &\text{s.t. } \sum_{i=1}^{k} \boldsymbol{A_{0i}} \boldsymbol{x_i} \leq \boldsymbol{b_0}, \\ &\quad \boldsymbol{A_{ii}} \boldsymbol{x_i} \leq \boldsymbol{b_i}, && i \in 1 \ldots k, \\ &\quad \boldsymbol{x} \in \boldsymbol{X}, \end{aligned}$$

where the first constraint defines the linking row or rows. One method to solve row linked problems is *Lagrangian relaxation*, which is the dual of Benders decomposition. Lagrangian relaxation relaxes the linking rows, by omitting them from the problem. As in Benders decomposition the resulting LP decomposes into several smaller problems which are comparatively easy to solve. As they have been omitted a method is needed to find the optimal value of the

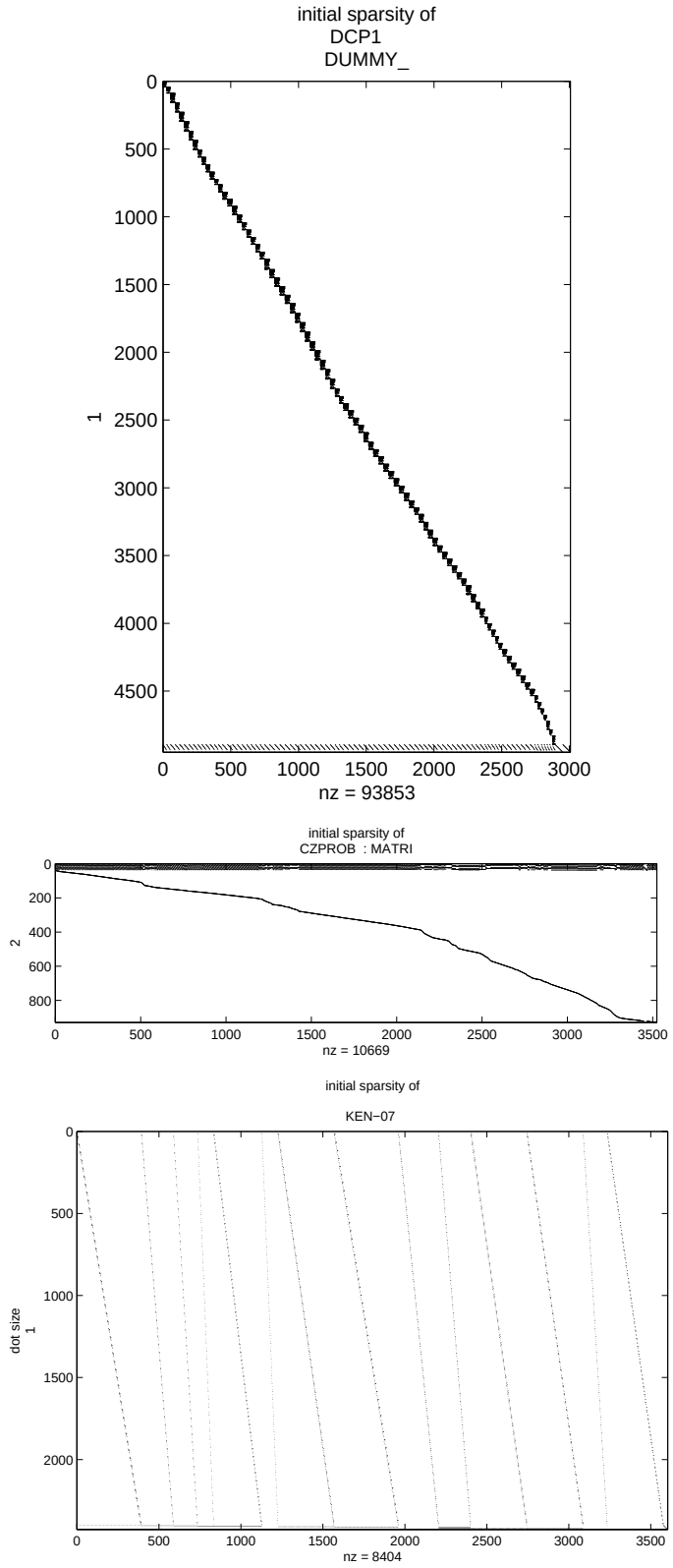


Figure 3.5: Row linked problems. DCP1 with linking rows 4895-4950, CZ-PROB with linking rows 1-7 and KEN-07 with linking rows 2402-2426

linking rows. This is done by adding costs on the variables in the rows into the objective of each subproblem. These extra costs can be updated by *subgradients*, by the *dual cutting plane method*, or by using an *augmented Lagrangian* framework. Lagrangian relaxation is introduced in more detail in Section 6.2.2 of Chapter 6.

The methods introduced in Sections 3.3.1 and 3.3.2 do not omit the linking rows, but instead modify the pivot row selection rule. The method of Section 3.3.3 omits the linking rows, then reinstates them. These three methods are tested on the large row linked problem DCP1. The method of Section 3.3.4 also omits the linking rows, and after some work reinstates them. It is tested on the row linked problems CZPROB and KEN-07. As with our methods for column linked problems, using these methods on such a small number of test problems only gives a basic test of the methods. In section 3.4 we therefore consider solving the whole Subnet set of problems.

3.3.1 Avoiding linking rows

This method is similar to the method of avoiding linking columns, introduced in Section 3.2.1 above. In the method of avoiding linking rows we alter the pivot row selection rule to only pivot on linking rows when there are no other attractive rows. Results for this method and two other methods are presented below, after all three methods have been introduced.

3.3.2 Avoiding dense rows

This method is similar to the method of avoiding dense columns, introduced in Section 3.2.2 above. In the method of avoiding dense rows the pivot row selection rule is adjusted to bias against choosing dense rows. In order to do this, the number of nonzeros in each row of the tableau, called the *row nonzeros* and denoted $\boldsymbol{z^{row}}$, must be known each iteration. $\boldsymbol{z^{row}}$ is calculated from the initial tableau then updated each iteration, which can be done cheaply when using the tableau simplex method. We solve DCP1 with $\nu^{row} = 1$.

3.3.3 Omitting rows

This method omits the linking rows, resulting in several smaller problems. Solving all of these is much easier than solving the original problem, and results in an approximate solution to the original problem. The linking rows are then reinstated and solved using the approximate solution as a warm start point. We can think of this method as similar to the method of avoiding linking rows, with the difference being that here once the linking rows are reinstated they are all available to be pivoted on.

Table 3.4 shows results for solving DCP1. We compare the standard tableau simplex solve with the three alternative row selection rules that were introduced in Sections 3.3.1, 3.3.2 and 3.3.3. Two of the methods, avoiding linking rows and omitting rows, were successful in reducing

	DCP1	
	pivots	work
standard solve	3418	1124M
avoiding linking rows	3282	896M
avoid dense rows	3348	1205M
omitting rows	3326	838M

Table 3.4: Results for different row selection rules

the amount of work, with little effect on the pivots. Avoiding dense rows was not successful on this problem.

The method of avoiding dense rows can be applied to all problems, not just those that are row linked. So although the method was not successful on DCP1, it is successfully applied to all Subnet problems in Section 3.4.3.

3.3.4 Omitting rows and using heuristic

In the method of omitting rows, described above, the problem is solved with all the linking rows omitted, then resolved with all the linking rows reinstated. Here we omit the linking rows and solve a series of LPs with the aim of finding a close to feasible point. The linking rows are then all reinstated and the original problem solved, warm starting from the close to feasible point.

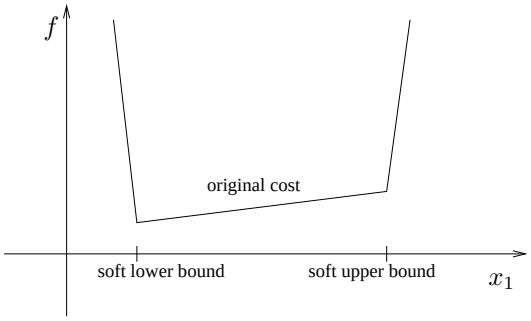


Figure 3.6: Response curve with soft bounds

Our method is a generalisation of the *primal allocation heuristic* of McBride [1998], which is suitable only for LPs such as *multi-commodity network flow problems*, where the variables in the linking rows all have positive coefficients and all have lower bounds of zero. An outline of the algorithm is given below.

1 LP solved without linking rows.
2 For all linking rows that are violated upper bounds on variables in those rows proportionally reduced.
3 LP resolved.

In Step 1 the LP is solved without the linking rows. After this the linking rows are typically violated. In Step 2 the upper bounds on variables in the linking rows are reduced, and in Step 3 a new LP is then solved. Step 2 and Step 3 continue repeatedly, gradually forcing feasibility in the linking rows.

The main part of the algorithm is Step 2, where bounds are changed. So that the new upper bounds don't make the LP infeasible they are soft bounds, meaning that the LP can violate them, albeit at the cost of a large objective value penalty. Figure 3.6 shows the response curve of a variable, x_1, with soft bounds. Consider an example of the primal allocation heuristic applied to a problem with just one linking row:

$$\begin{aligned} \text{linking row: } & x_1 + 2x_2 + 3x_3 \leq 1, \\ \text{variable values: } & \hat{x}_1 = 0, \quad \hat{x}_2 = 0.5, \quad \hat{x}_3 = 1, \\ \text{row activity: } & 4, \\ \text{variable bounds: } & 0 \leq x_i \leq 1, \ i \in 1 \ldots 3. \end{aligned} \tag{3.4}$$

In this example the row activity of the linking row is four times the upper bound. To force the row activity down the upper bounds on the variables in the linking row are reduced. Each nonzero variable is forced down by a factor of four, i.e given a new upper bound of one quarter its current value. A feature of the algorithm is that feasibility is not sought in one iteration, so not all variables have their bounds adjusted. In particular, variables at their lower bound, such as x_1, do not have their bounds changed. Thus the problem is resolved with new bounds. In our example the new bounds are:

$$\text{new bounds: } 0 \leq x_1 \leq 1, \quad 0 \leq x_2 \leq 0.125, \quad 0 \leq x_3 \leq 0.25. \tag{3.5}$$

Because x_1 has not had its bounds changed the next solution may also violate the linking row, in which case tighter bounds are enforced. This continues until a near feasible point is reached. The example above can be written generally as:

$$\begin{aligned} \text{linking row: } & \boldsymbol{A}^T \boldsymbol{x} \leq b, \\ \text{row activity: } & \boldsymbol{A}^T \hat{\boldsymbol{x}} = \hat{b}, \\ \text{variable bounds: } & \boldsymbol{0} \leq \boldsymbol{x} \leq \overline{\boldsymbol{x}}, \end{aligned} \tag{3.6}$$

with $\boldsymbol{A}$ a vector of positive constants, b and $\overline{\boldsymbol{x}}$ upper bounds on the row and variables respectively and $\hat{\boldsymbol{x}}$ the current solution and $\hat{b}$ the current row activity. If the row activity exceeds the upper bound then any variable i with $\hat{x}_i > 0$ gets a new upper bound:

$$\text{new bound: } \min(\overline{x}_i, \hat{x}_i \frac{b}{\hat{b}}). \tag{3.7}$$

Thus the primal allocation heuristic reduces the feasible space of the $\boldsymbol{x}$ variables in towards the origin. The primal allocation heuristic is not suitable for more general row linked problems, as they can have linking rows containing variables with negative coefficients and nonzero lower bounds. We therefore extend the method to form a new method that we call the *extended primal allocation heuristic*, which attempts to force feasibility of the linking rows by adjusting both lower bounds and upper bounds. The basic algorithm is the same, but we change how bound are reduced in Step 2. Consider this example, also with only one linking row:

$$\begin{aligned} \text{linking row: } & 0 \leq x_1 - 2x_2 + 3x_3 \leq 1, \\ \text{variable values: } & \hat{x}_1 = 0, \quad \hat{x}_2 = 0.5, \quad \hat{x}_3 = 1, \\ \text{row activity: } & 2, \\ \text{variable bounds: } & 0 \leq x_i \leq 1, \ i \in 1 \ldots 3. \end{aligned} \tag{3.8}$$

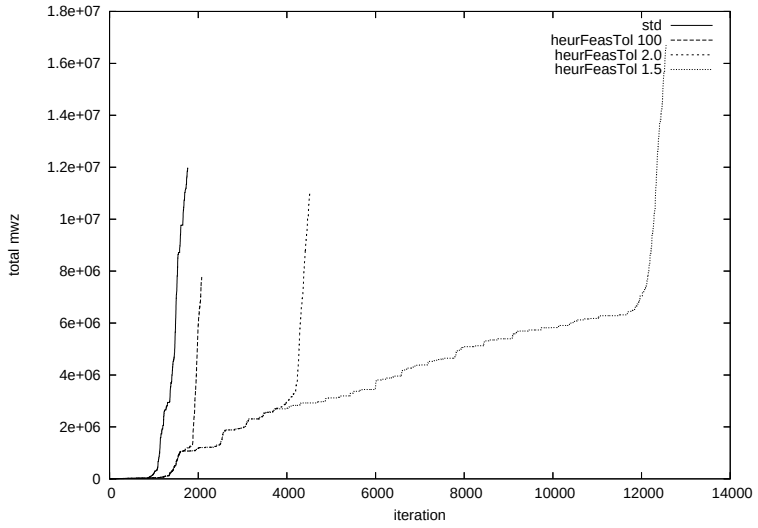
In the example above the variables in the linking row have a mixture of positive and negative coefficients. The row activity is above the upper bound, so the variable bounds must be changed. As in the primal allocation heuristic we do not change bounds on any variable that is at an appropriate bound, such as x_1. In the extended primal allocation heuristic lower bounds and upper bounds can be moved. As the row activity is twice its upper bound we move an appropriate bound on each of x_2 and x_3 halfway between the variable value and the current bound. This means increasing the lower bound of x_2, and lowering the upper bound on x_3.

$$\text{new bounds: } 0 \leq x_1 \leq 1, \quad 0.25 \leq x_2 \leq 1, \quad 0 \leq x_3 \leq 0.5. \tag{3.9}$$

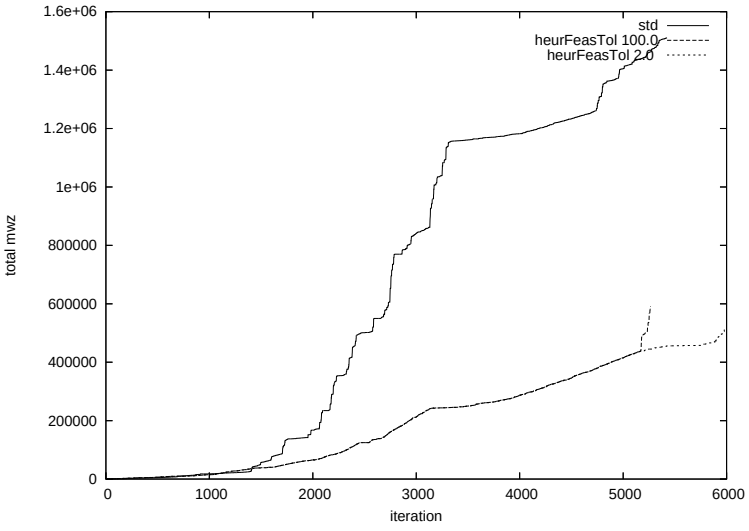
In general, with the extended primal allocation heuristic, if the linking row activity is above its upper bound (as in the example above), then a variable with a positive coefficient in the row has its upper bound decreased, and a variable with a negative coefficient in the row has its lower bound increased, and vice versa if the linking row activity is below its lower bound. As before these bounds cannot be relaxed, but are soft. If the row activity is zero, leading to dividing by zero when calculating the new bounds, we perturb the row slightly.

In the two examples above there was only one linking row. When there are several linking rows there is the potential difficulty that each row may suggest different alterations to a variable's bounds. Indeed, it is possible that the lower bound for a particular variable suggested by

one violated linking row is above the upper bound suggested by another violated linking row. In the extended primal allocation heuristic, after exploring different options, it was decided to resolve this difficulty by updating the variable bounds using the average bound movement suggested by all the violated linking rows. The algorithm continues until the linking rows are all nearly feasible. We use a measure of feasibility that is approximately equal to the number of linking rows that are violated, with rows that are nearly satisfied contributing less than rows that are far from satisfied. When this measure falls below the parameter $heurFeasTol$ the linking rows are reinstated. Selecting an appropriate value of $heurFeasTol$ is a difficulty of the method.



(a)



(b)

Figure 3.7: Cumulative work per iteration on standard solve, and with extended primal allocation heuristic, on CZPROB (a) and KEN-07 (b)

The extended primal allocation heuristic was tested on the row-linked problems CZPROB and KEN-07. In these tests we used a simplified version of the soft bounds, with a maximum of two piecewise linear cost sections per variable (instead of three). Figure 3.7 shows the cumulative amount of work for the standard solve and using the extended primal allocation heuristic with different values of $heurFeasTol$. The sharp upturn in the amount of work for each solve occurs when the heuristic reinstates the linking rows and switches to the normal simplex method. For CZPROB, using the heuristic adds greatly to the number of iterations, though of course these extra iterations are on an LP without the linking rows so are cheap to perform. The most efficient solve is with $heurFeasTol = 100$. With such a high value of $heurFeasTol$ this solve immediately reinstates the linking rows after the first LP optimisation, so is in fact equivalent to the method of omitting rows. For KEN-07, the most efficient solve is with $heurFeasTol = 1.5$. With this small value of $heurFeasTol$ an approximate solution is found that is almost feasible. This solve had significantly less work than the standard solve.

Although it is successful on the two test problems, this method is not applicable to general LPs. A potential improvement to the extended primal allocation heuristic is possible for row linked block structured problems, such as KEN-07 (whose block structure is not visible without a permutation of the tableau) and DCP1. This is to add extra *block usage* variables to the problem, that represent the contribution to each linking row by each block. The heuristic could then alter the bounds on this small set of block usage variables, rather than on every variable that appears in the linking rows. This has not yet been implemented.

3.4 Solving Subnet problems

We now consider solving all the Subnet problems. Of the methods introduced in the previous sections only the methods of avoiding dense columns and avoiding dense rows are applicable to problems which are not row and column linked, so those are the only two of the previous methods that we try on all Subnet problems. In Section 3.4.1 we look at altering the pivot row selection rule to avoid dense rows, in Section 3.4.2 at altering the pivot column selection rule to avoid dense rows, and in Section 3.4.3 at altering them both. In Section 3.4.4 we look at how these alterations of the pivot selection rule can be applied to the revised simplex method. In Section 3.4.5 we look at exploiting the Tarjan form to give a new method to solve LPs that preserves sparsity.

3.4.1 Pivot row choice

The standard pivot row selection rule used in Simplex 10 was described in Section 2.1.2. In summary, we select the row that minimises the sum of infeasibilities, subject to not increasing the number of infeasibilities. Apart from the sum of infeasibilities the only other important row

property was found to be the size of the pivot element, therefore in Simplex10 we tiebreak on two otherwise equally good rows by choosing the row with the larger pivot element. In Section 3.3.2 we weighted row attractiveness by sparsity of the row, which was unsuccessful on DCP1.

We now consider using sparsity of the row as a tiebreak between otherwise equally good rows, and compare this with tiebreaking on pivot size. Note this this is just a tiebreak on pivot row choice using sparsity, and is not the same as picking sparse rows. Picking sparse rows was done on DCP1 in Section 3.3.2, and will be done on all the Subnet problems in Section 3.4.3.

Solve	row selection	pivot ratio	std.dev	work ratio	std.dev
0	pivot size tiebreak	1	0	1	0
1	sparsity tiebreak	0.971	0.028	0.877	0.087
2	sparsity tiebreak	0.988	0.033	0.960	0.113
3	sparsity and pivot size tiebreak	1.101	0.171	1.574	0.811

In the table above Solve 0 is the standard solve, where we tiebreak by pivot size. We choose to compare solves by work rather than time. Solving all of the Subnet problems by Simplex10 takes around one hour.

In Solve 1 we tiebreak on row choice by row sparsity. In Solve 2 we also tiebreak on row choice by row sparsity, but alter the other row parameters so that a great many rows have the same score, so that the tiebreak by sparsity has a more pronounced effect. In Solve 3 we tiebreak on row choice by both row sparsity and pivot size, also with the parameters that mean a great number of rows are given an equal score before the tiebreak.

The final four columns of the table compare the average number of pivots and amount of work of a solve to the standard solve. To generate the pivot ratio for a solve we first calculate a ratio for each problem of pivots in the solve to pivots in the standard solve, then take the weighted average of these ratios, weighting by number of pivots in the standard solve of each problem. A similar method is used to find the work ratio (in which we also weight the problems by number of pivots in standard solve). A pivot ratio or work ratio of less than one is an improvement on the standard solve. The standard deviations of the ratios are also given.

The table shows that both Solve 1 and Solve 2 reduce the average amount of work, and surprisingly also reduce the average number of pivots. Solve 1, where the tiebreak is only among a small row choice, is the better of the two. In Solve 3, where we try to combine the benefits of a large pivot size and a sparse row, the number of pivots and amount of work increases, with high standard deviation on both measures. It is possible though that with fine tuning of parameters and safeguards against very bad pivots being chosen, tiebreaking on rows for both sparsity and pivot size could work well.

Solves 1-3 (and Solves 5 and 7 below) were repeated with different solve parameters that increased the average number of rows that were attractive at each iteration, in the hope that a greater row choice would mean better rows could be chosen. However this simply increased

the number of pivots, and did not lead to less work.

3.4.2 Pivot column choice

Solve	pivot selection rule	pivot ratio	std.dev	work ratio	std.dev
4	column by sparsity	1.101	0.051	0.797	0.073

We saw in Section 3.2.2 that selecting pivot columns by their sparsity is effective. In Solve 4 this method of avoiding dense columns is applied to all the Subnet problems. Although the average number of pivots is increased in Solve 4 the amount of work is significantly less than the standard solve.

3.4.3 Pivot column and row choice

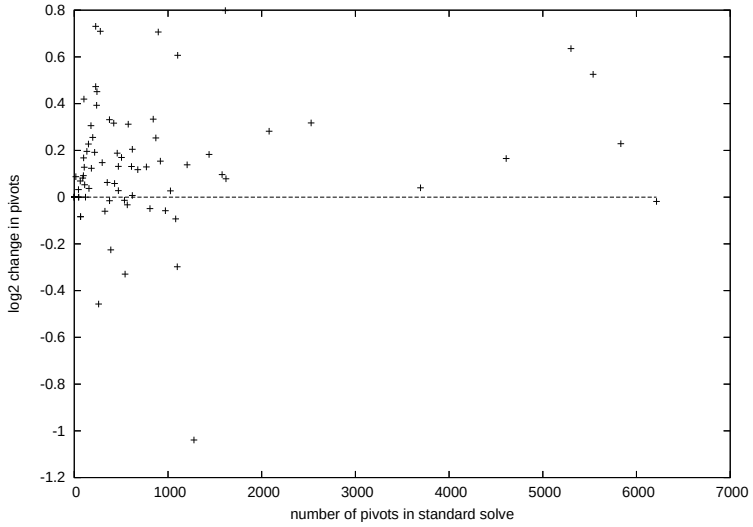
Solve	pivot selection rule	pivot ratio	std.dev	work ratio	std.dev
5	column and row by sparsity	1.1214	0.050	0.602	0.079

We now consider picking both sparse columns and rows. In Solve 5 at each iteration of the simplex method we first choose a pivot column weighting column attractiveness by sparsity, as in Solve 4, then choose a pivot row weighting row attractiveness by sparsity, as in Section 3.3.2. This method of row selection was found to be more effective than merely tiebreaking on row sparsity, as in Solves 1-3 above. Solve 5 has more pivots than the standard solve but about 40% less work.

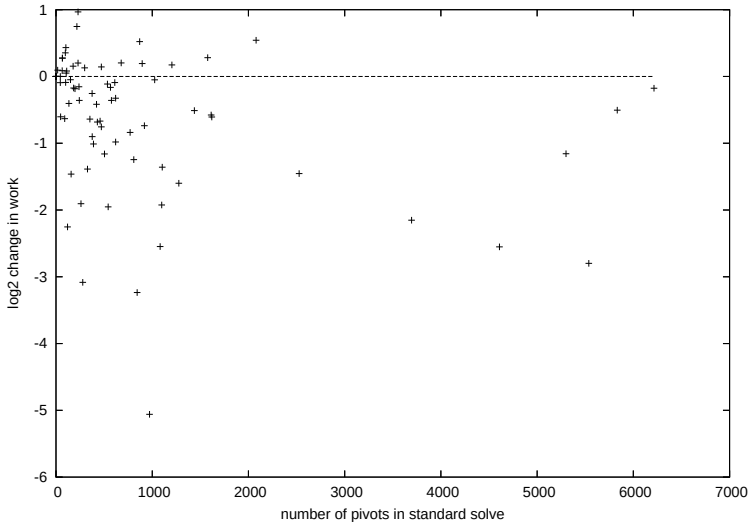
Figure 3.8 shows the change in the number of pivots and amount of work using Solve 5 compared to the standard solve. On both graphs the x-axis is the number of pivots in the standard solve (a good indication of the problem size) and the y axis the logarithm to the base two of pivots or work compared to the standard solve. Any value below 0 indicates an improvement on the standard solve, and a value of -1 for example indicates a problem took half as many pivots (or half as much work).

3.4.4 Pivot column and row choice in the revised simplex method

In this section we attempt to adapt the methods of picking a sparse pivot column and a sparse pivot row (Solve 5 above) for use with the revised simplex method. As described above these methods requires explicit knowledge of the tableau each iteration, as we must know the number of row nonzeros ($\boldsymbol{z^{col}}$), and the number of column nonzeros ($\boldsymbol{z^{row}}$). This information is not available in the revised simplex method. To adapt Solve 5 for use with the revised simplex method we must therefore estimate $\boldsymbol{z^{col}}$ and $\boldsymbol{z^{row}}$. We assume that we start our solve with an



(a)



(b)

Figure 3.8: Solve 5 pivots for each problem (a) and work for each problem (b), against number of pivots to solve in standard run

all logical basis, and the initial row and column nonzeros are known. We update the estimates of row and column nonzeros each iteration. We describe two possible ways to do this, of which the second way is preferred.

Firstly, we can directly update the estimates using information from each pivot. We call this the *direct method*. In the revised simplex method although the whole tableau is never generated, the pivot column and row are generated each iteration, and can be used to give exact values for the number of nonzeros in that column and row. Using the position of the nonzeros in the pivot column and row we can estimate where fill in will occur in other rows

and columns, and update all the estimates. We must also scale the increase in the estimates to allow for numerical cancellation and superposition at levels other than what is expected from a uniformly random tableau. The direct method works fairly well as long as each pivot adds a structural to the basis in place of a logical, and so increases the density. However, if a pivot removes a structural from the basis then *structural cancellation* occurs, and the tableau can actually become less dense. Structural cancellation will not be detected by the direct method.

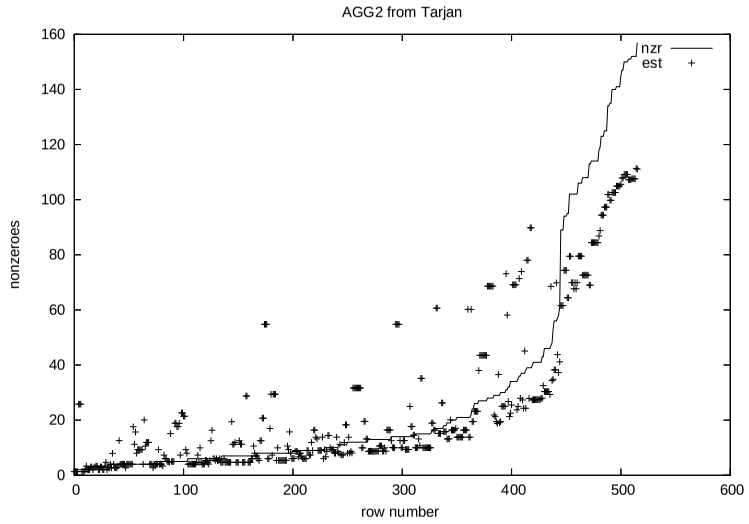
It is because of this difficulty that we introduce what we call the *indirect method*. In the indirect method at each iteration of the simplex method we generate a minimal list of pivots that lead from the initial, all logical basis, to the current basis. This list can, for example, be generated by getting the Tarjan form of the current basis. This minimal list of pivots we call the *Tarjan pivots*, to distinguish them from the *simplex pivots* which are the pivots actually used to solve the problem. All the Tarjan pivots are of the type that add a structural to the basis and remove a logical, in which there is no structural cancellation. We can therefore get row and column nonzero estimates by applying the list of Tarjan pivots to a new copy of the initial tableau, updating the estimates each Tarjan pivot as in the direct method. We can still use the exact values of number of nonzeros in the pivot row and pivot column that came from the most recent simplex pivot. We get exact values from some previous simplex pivots (including the most recent), as long as they are for rows and columns that have not experienced any fill in since they last featured in a simplex pivot.

Whether we use the indirect or direct method a great difficulty is predicting the amount of numerical cancellation and superposition to allow for when updating the estimates. Without access to the tableau we cannot hope to know exactly where numerical cancellation and superposition occur. Suppose we call the average rate of numerical cancellation K_1, and the average rate of fill in compared to what is expected from a uniformly random tableau K_2. We could try to calculate K_1 by looking at the numerical values of the initial tableau. We could try to calculate K_2 by looking at the structure initial tableau. Depending on the structure of a problem K_2 could be greater or less than one, though considering the findings of Chapter 2, in particular the results in Figure 2.10, we expect it to be less than one for almost all Subnet problems. This is because real problems have more superposition than would be expected with a random scattering of nonzeros in the tableau.

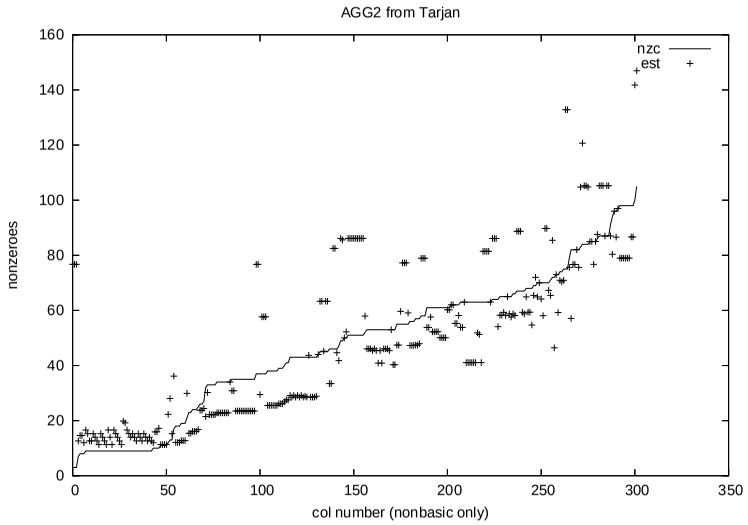
As both K_1 and K_2 simply affect the increase in the row and column nonzero estimates it is only their combined effect that is needed. Therefore rather than trying to calculate K_1 and K_2 themselves, we instead estimate the cumulative effect of K_1 and K_2 after all iterations. This is done by comparing exact values of number of row and column nonzeros, which are known for some recent simplex pivot rows and columns, with our estimates for those rows and columns. This gives a scaling factor which approximates the total effect of K_1 and K_2. This scaling factor is then applied to all rows and columns.

The formula used for estimating the increase in row and column number of nonzeros is given

in Section C of the appendix.



(a)



(b)

Figure 3.9: Estimates of row (a) and column (b) number of nonzeros

Figure 3.9 compares the actual row and column number of nonzeros (labelled nzr and nzc) with our estimates, at the solution to Hyper-sparse problem AGG (number 38). The graphs are ordered by actual row or column number of nonzeros.

Solve	pivot selection rule	pivot ratio	std.dev	work ratio	std.dev
6	column and row by sparsity	1.047	0.043	0.896	0.067

In Solve 6 we solve the Subnet problems choosing what are believed to be sparse rows

and columns, using the estimates of row and column number of nonzeros. This is an attempt to apply the successful Solve 5 in a way that can be used with the revised simplex method. Rather than measure work as the sum over all iterations of the product of pivot row nonzeros and pivot column nonzeros, as we did for the previous solves using the tableau simplex method, we measure work as the sum over all iterations of the addition of the pivot row number of nonzeros and pivot column number of nonzeros. This is because in the revised simplex method the major effort each iteration is in forming the pivot column, which requires work proportional to the number of nonzeros in the column, and in forming the pivot row, which requires work proportional to the number of nonzeros in the row.

Solve 6 solves with more pivots but less work than the standard solve, although the saving in work is less significant than in Solve 5. The work given for Solve 6 does not include the work involved in finding the row and column estimates, which can be significant.

3.4.5 Exploiting Tarjan form

In this section we introduce a new method for selecting both the pivot column and row, that could be applied directly to the revised simplex method. We presume throughout the section we have access to the basis in Tarjan form and the permuted nonbasis.

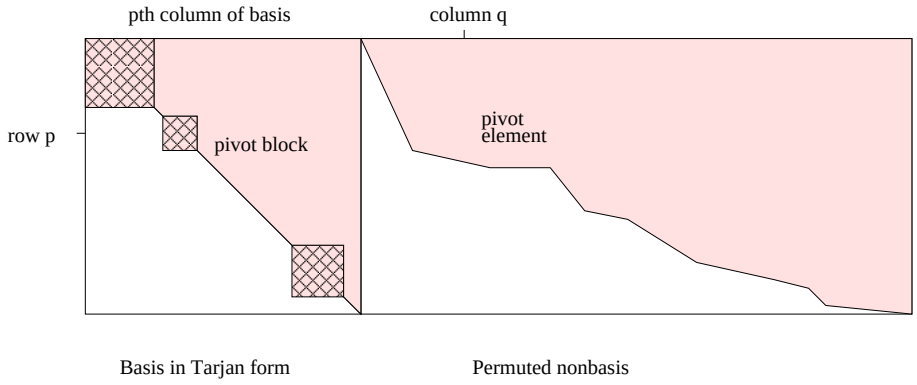


Figure 3.10: Pivot at row p and column q

Figure 3.10 shows an example basis in Tarjan form and permuted nonbasis. This form can also be seen for the optimal solution to Hyper-sparse problem LOTFI in Figure 2.18. At a given iteration we denote the pivot row as row p and the pivot column as column q of the permuted nonbasis. The pivot makes the p^{th} column of the basis becomes nonbasic, as it is swapped with column q. Thus the choice of pivot column and row can be thought of as the choice of which column should enter and leave the basis, respectively. We call the (possibly trivial) diagonal block that contains the p^{th} column of the basis the *pivot block*.

We saw above (Figures 2.15 and 2.16) that with a basis in Tarjan form the density of the basis inverse, and hence the tableau, is heavily affected by the size of the diagonal blocks. To preserve sparsity in the tableau we can therefore try and pick pivot rows and columns that

minimise the size and number of diagonal blocks. We consider the choice of the entering column to the basis (the pivot column), then the choice of the leaving column, which is determined by the pivot row.

Entering Column

We choose an entering column q with the intention of forming no new diagonal blocks in the Tarjan form of the basis.

The pivot column q always has an entry at position p. There are then three cases to consider. The first case is if column q only has its other entries above row p. In this case the column breaks up the pivot block, though the remaining columns of the block may still form a strong component. The second case is if column q has some other entries above and below row p, but with those below row p within the pivot block. Here the block structure of the basis is not worsened, and may even be improved. The third case is if column q has some other entries below the furthest extent of the pivot block where it enters the basis. We call this an *overhang*. Here the block structure of the basis may be worsened, by linking the pivot block to other blocks. To be sure that the entering column does not worsen the block structure it must therefore be a column that has no overhang. However, the pivot block is determined by the pivot row, which is not known when the pivot column is chosen. Therefore when picking the pivot column we cannot be sure whether or not there will be an overhang.

It is clear though that it is best to pick an entering column as far left as possible from the permuted nonbasis, to give the best chance of being in one of the first two cases. We therefore introduce a column selection rule that favours columns from the leftmost side of the nonbasis. We bias the column choice by altering the score of each column, in a similar manner to what was used in avoiding dense columns and dense rows (see (3.2)). The column score s_i for row i is altered to give:

$$s'_i = s_i(q_i)^{-\nu^{pos}}, \tag{3.10}$$

where q_i is the position of column i in the permuted nonbasis and ν^{pos} a parameter (typically equal to one). To use this formula the permuted basis (which is cheap to find once the Tarjan form is known) must be calculated every iteration. If this is not done then old values of $\boldsymbol{q}$ can be used, and every column which has become nonbasic since the last time the permuted nonbasis was calculated is given the mean value of $\boldsymbol{q}$.

Leaving column

The choice of leaving column can help break up any existing block structure.

Replacing a column from a diagonal block is desirable as it can break up the block structure. The best columns to remove from the basis matrix are those on the right hand side. These are often the columns with the most nonzeros, but more importantly when these columns are replaced with the entering column it is unlikely new blocks will be formed as there is less likely to be a significant overhang. We therefore weight the pivot row selection using a similar formula to the pivot column selection in (3.10), to bias towards pivot rows that correspond to columns on the right hand side of the basis. The rule is also adjusted to account for the fact that any columns from within the same diagonal block, regardless of their position within the block, are equally preferable to remove from the basis. As with the column selection rule out of date information can be used if the Tarjan form is not available every iteration.

We have a new column selection rule, based on the analysis of the permuted nonbasis, and a new row selection rule, based on the analysis of the Tarjan form. The intention of both rules is to minimise the size and number of diagonal blocks in the Tarjan form of the basis, as diagonal blocks are known to lead to dense basis inverses and dense tableaux.

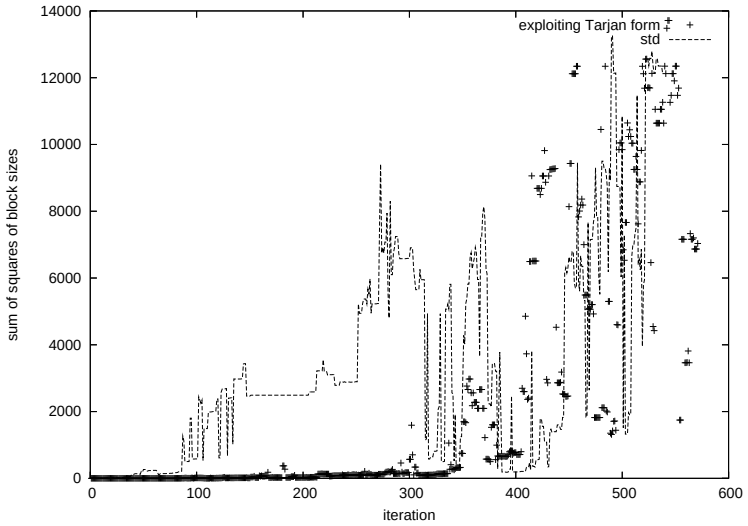
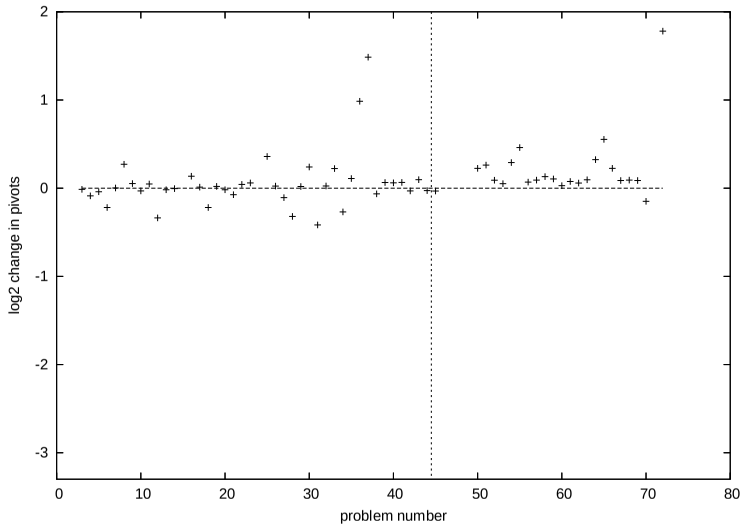


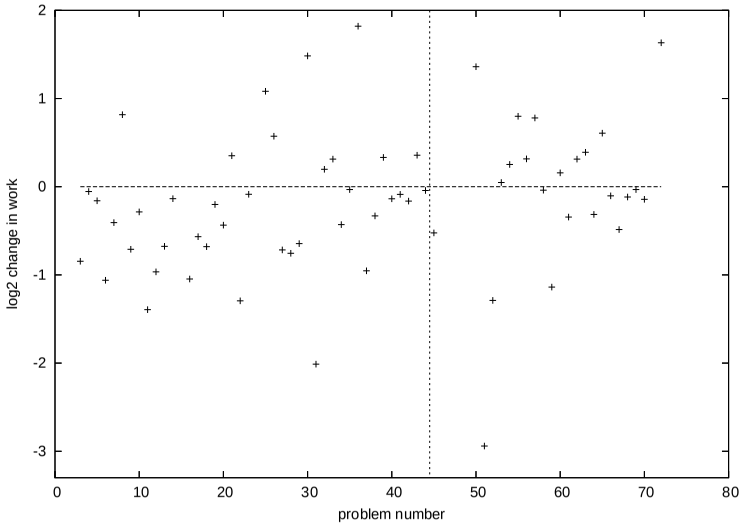
Figure 3.11: Effect of new column selection rule

We can think of the column selection rule as intending to prevent diagonal blocks from forming, and the row selection rule as intending to break up any blocks that have formed. As an example of the potential effectiveness of the column selection rule consider Figure 3.11. It shows the size of the diagonal blocks, with and without selecting pivot columns that exploit the Tarjan form. The new rule is successful at preventing large blocks being formed for the first 300 or so iterations, though large blocks do then appear. For some problems, such as Hyper-spare problem STANDMPS (number 17), with this new column selection rule the problem solves without ever forming any diagonal blocks.

Solve	pivot selection rule	pivot ratio	std.dev	work ratio	std.dev
7	column and row exploiting Tarjan form	1.102	0.072	0.936	0.105



(a)



(b)

Figure 3.12: Solve 7 pivots for each problem (a) and work for each problem (c)

Results

We now apply both the new column and row selection rules, in Solve 7. The table shows that Solve 7 is able to reduce the work slightly compared to the standard solve. The work required to get the Tarjan form and permuted nonbasis is not included. Figures 3.12(a) and 3.12(b) show the change in number of pivots and work respectively for each problem with Solve 7. The x-axis

is the problem number and the as in Figures 3.8(a) and 3.8(b) the y axis is the logarithm to the base two of pivots or work compared to the standard solve. Any value below 0 indicates an improvement on the standard solve, and a value of -1 for example indicates a problem took half as many pivots (or half as much work). In general the Hyper-sparse problems take around the same number of pivots as in the standard solve, and less work, though this varies greatly from problem to problem. The Densefill problems take more pivots than the standard solve, and about the same work, though again this varies greatly from problem to problem. One unusual result is Densefill problem DEGEN2 (number 51), which takes 7.7 times less work. Overall the method is shown to be effective, in particular for Hyper-sparse problems.

Solve 7 was repeated only calculating the Tarjan form and permuted nonbasis every five iterations, so out of date information had to be used in the pivot column and pivot row selection. This had a highly varied effect on different problems, but was generally worse than calculating the Tarjan form and permuted nonbasis every iteration, and about equal with the standard tableau simplex solve.

We now briefly consider four other ways that the Tarjan form could be used to influence pivot row and column selection. Firstly we could try and pick a pivot row that is especially suitable for the given entering column, for example picking a pivot row so that the entering column enters into a diagonal block and has no entries below that diagonal block. Secondly it may be beneficial to bias row selection to select leaving columns that have many entries close to the diagonal, as these are known to make the inverse denser (Observation 1). Thirdly we could use the rule that when the basis is viewed as a graph each diagonal block is a set of interconnected nodes, and every block consists of some forward and some backward arcs. If a block has only very few backward arcs (say) then removing the columns with backward arcs could be prioritised. Fourthly we can evaluate the general importance of an arc in forming a strong component by seeing how many closed paths it appears on. This can be done by solving a maximum flow problem, sending flow round the graph from each node back to itself. Arcs that are identified as appearing on many closed paths could be targeted for removal.

All the work in this section has considered a basis matrix in Tarjan form. Using a different way of representing the basis, such as the *Tomlin form*, could lead to further methods for choosing columns to enter or leave the basis.

3.5 Summary

In this section we have introduced several new methods for solving column and row linked problems, then methods for solving all the Subnet problems.

On the column linked problems, the methods of avoiding linking columns, avoiding dense columns and pushing columns all reduced the work by at least 20% on at least one of the two test problems. On the row linked problem DCP1 avoiding linking rows and omitting rows reduced

the work by about 20% and 25% respectively. The primal feasibility heuristic of McBride and Mamer was extended to general row linked problems. The new heuristic, with the best value of $heurFeasTol$ for each problem, reduced the total work by 35% and 65% for the row linked problems CZPROB and KEN-07 respectively. Although these methods for row and column linked problems show potential they have not been tested on a large number of problems.

When solving all the Subnet problems tiebreaking on pivot row choice by row sparsity (Solve 1) saved work. Trying to select both sparse columns and rows (Solve 5) reduced the total work by an average of about 40%. For the tableau simplex method, this is the recommended algorithm.

A method was developed to estimate the row and column number of nonzeros each iteration, for use with the revised simplex method. Using these estimates, instead of the actual row and column number of nonzeros, Solve 5 was repeated as Solve 6, and now gave a 10% reduction in work, ignoring the work required to estimate the row and column number of nonzeros. Finally we developed new pivot column and row selection rules based on putting the basis in Tarjan form and forming the permuted nonbasis (Solve 7). This solve also saved work, ignoring the work required to get the Tarjan form, and can be easily adapted to the revised simplex method. Solve 6 and Solve 7 were not as successful as Solve 5, but may be able to be successfully incorporated into a revised simplex method solver.

3.5.1 Further work

The methods for solving row and column linked problems could be tested on several other suitable problems, and adapted so that they can be used in the revised simplex method.

The methods for solving all the Subnet problems could be directly incorporated into a revised simplex algorithm, to try to realise some of the gains in work suggested.

Chapter 4

Diet Problems

The following four chapters are about *diet problems*. In this chapter we introduce the *blending problem*, the *pooling problem* and the *generalised pooling problem* (GPP). The GPP is a fairly new problem. The company Format International solve diet problems on behalf of various clients. They are interested in being able to quickly solve the GPP and its variants. In Chapter 5 we look at solving the GPP, in Chapter 6 at a multifactory GPP and in Chapter 7 at solving a GPP with the addition of *silos* to the model, which add integer constraints.

In this chapter we introduce a new formulation for the pooling problem, called the extended q-formulation, which we use to model GPPs. We consider some relaxations of the new formulation on a test set of GPPs (and consider other, linear, relaxations in Chapter 5). Based on observations of Grothey et al. [1999], we examine the pooling problem as the geometric problem of where to place the mixing bins in the nutrient space, and this new analysis shows how the mixing bins affect the feasibility and objective of a pooling problem.

4.1 Introduction

The most basic diet problem is the blending problem. In a blending problem raw materials, often called simply raws, are combined to form a product. Each raw has a known cost and known concentrations of each of several nutrients. The *nutrient concentration* is the amount of the nutrient, in whatever units are relevant for that nutrient (grams, joules, etc.), per tonne of the raw. The product has a given mass and a range of acceptable concentrations for each nutrient.

Solving the problem means finding the mass of each raw that should be combined to make the product at minimum cost. The first blending problem was posed in Stigler [1945], and shortly afterwards was modelled and solved as the first LP by Dantzig.

$$\min_{\boldsymbol{t},\boldsymbol{z}} \sum_{r=1}^{R} C_r t_r \tag{4.1}$$

$$\text{s.t.} \sum_{r=1}^{R} X_{rn} t_r = z_n \qquad \forall n \in 1 \ldots N, \tag{4.2}$$

$$\underline{z}_n \leq z_n \leq \overline{z}_n \qquad \forall n \in 1 \ldots N, \tag{4.3}$$

$$t_r \geq 0 \qquad \forall r \in 1 \ldots R. \tag{4.4}$$

The mathematical model of the blending problem is given above. There are R raws, indexed with r, and N nutrients, indexed with n. The parameters of the model are C_r, the cost of raw r, X_{rn}, the nutrient concentration of nutrient n in raw r, and $\underline{z}$ and $\overline{z}$, the lower and upper bounds on z_n. The variables are t_r, the mass of each raw r used, and z_n, the concentration of nutrient n in the product ($\boldsymbol{z}$ could be eliminated from the model but we prefer to leave it in).

The objective, (4.1), is to minimise the total cost, which is the sum of cost times mass for all raws. Constraint (4.2) defines the concentration of each nutrient in the product, which is the sum over all raws of nutrient concentration times mass. Constraint (4.3) bounds the concentration of each nutrient in the product. Constraint (4.4) ensures that a non-negative mass of each raw is used.

4.1.1 Size extensions

A blending problem may be enlarged by adding more raws or more nutrients. This is increasing the value of R or N in the model above. An example of enlarging a blending problem is Stigler's Diet Problem Revisited (Garille and Gass [2001]). Stigler's original diet problem had just nine nutrients. In the revisited problem (among other changes) the number of nutrients is increased to 31.

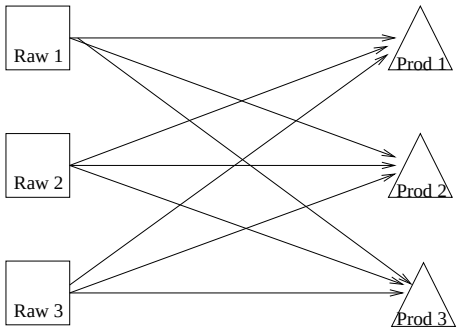


Figure 4.1: Network of possible flow of example diet problem

Other diet problems can be formed by extending the blending problem. It can be extended by making multiple products from the raws, combining multiple factories (each factory being one blending problem), or by modelling multiple time periods, with a blending problem solved

in each time period. Figure 4.1 shows the network of possible flow for an example diet problem which has multiple products. In a network like this possible flow between raws and products is represented by an arc between the raw and the product.

Models with extra products, factories or time periods require more variables and constraints than a simple blending problem, but are still LPs. They can lead to a model that is more accurate representation of the real situation. Williams and Redwood [1974] state that the ideal problem they would like to solve includes multiple products, multiple factories and multiple time periods. Solving an extended problem like that is advantageous compared to solving many small problems as the user does not have to assign resources between products/factories/time periods in a heuristic way. However, the extended problem is typically much larger and consequently harder to solve.

4.1.2 Difficulty extensions

Some real life situations require a model that is a nonlinear program (NLP) or even integer linear programs (ILP). Both NLPs and ILPs are much harder to solve than LPs. The extensions that make a problem an NLP or ILP arise naturally when modelling diet problems. All linear functions are *convex*, meaning that they have non-negative curvature throughout. Nonlinear functions may also be convex, or may be *nonconvex*. When they are nonconvex there is the possibility of having multiple *local minima*. The best of the local minima is called the *global minimum*. This is illustrated in Figure 4.2.

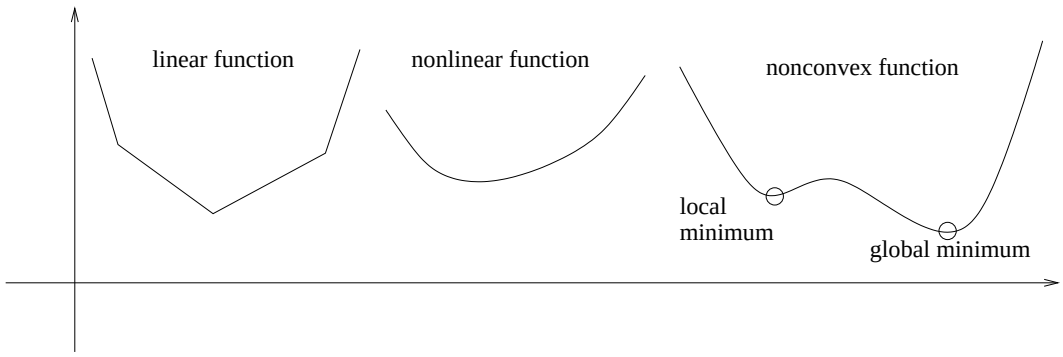


Figure 4.2: Example linear, nonlinear and nonconvex functions

Nonlinear programs can be formed from the blending problem in a variety of ways. One assumption made when solving the blending problem is that the nutrients in different raws combine additively, so that we can define $\boldsymbol{z}$ by constraint (4.2). If this assumption is not true constraint (4.2) must be replaced by a nonlinear constraint. A nutrient that often does not combine additively is metabolisable energy, in e.g. feed of broiler as described in Guevara [2004]. Similarly Hertzler et al. [1988] find that the energy requirement of cattle is nonlinear with respect to weight gain, leading to nonlinear objective functions that include weight gain terms as well as cost terms. In Munford [1996] the standard deviation of a particular nutrient

concentration is a variable, which depends on the product composition in a nonlinear way. This also requires a nonlinear constraint in the model.

In an integer program extra constraints are needed forcing some variables to take integer values. These programs can be formed from the blending problem if, for example, the mass of a particular raw must be a discrete value, or if the mass of a particular raw must be above some minimum amount if it is used at all. Logical conditions in the model (if-then constraints) also cause integer programs. All integer programs are nonconvex.

4.1.3 Pooling problem

In this section we introduce the pooling problem, which originated in Haverly [1978]. It can be thought of as a blending problem with both size and difficulty extensions. The size extension in the pooling problem is the introduction of multiple products. In Chapter 6 there is a further size extension to the pooling problem where we consider multiple factories. The difficulty extension in the pooling problem is the introduction of *mixing bins*, which can make the problem an NLP. In Chapter 7 there is a further difficulty extension to the pooling problem of adding *silos* with associated integer constraints, leading to a MINLP.

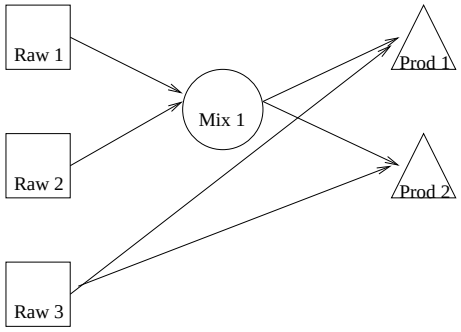


Figure 4.3: The first reported pooling problem, by Haverly [1978]

The mixing bins of the pooling problem allow raws to combine into intermediate mixes before combining again to form the products. Although strictly we think of a *mix* as the foodstuff and a *mixing bin* as the container holding the mix, we use the terms mix and mixing bin may be used more or less interchangeably. Figure 4.3 shows a pooling problem with one mixing bin. The mixing bin has inputs from two of the raws and output to both products. Thus the raws may combine in the mixing bin, and (identical) output can be sent to both products. Collectively raws, mixes and products are known as *bins*. In problems with mixing bins some raws may still flow directly into the products. These raws are known as *straights*. In Figure 4.3 there is one straight, Raw 3, flowing directly into Product 2. Pooling problems may have any number of nutrients, though Haverly's example problem of Figure 4.3 has only one.

Audet et al. [2004] showed that any mixing bins that do not have both multiple inputs and multiple outputs can be removed from the network, by merging them with another bin. A

pooling problem with no mixing bins is simply a blending problem. As pooling problems are nonlinear, and blending problems linear, it is the mixing bins that causes nonlinearity in the pooling problem. As well as being potentially nonlinear pooling problems are nonconvex (apart from trivial instances), meaning they can have multiple local minima.

There are several different formulations used to model the pooling problem. The formulations each lead to a different mathematical model of the problem. These models are all equivalent though, in that they all have the same global optimal solution. Haverly used what has become known as the p-formulation. Since then the q-formulation has been introduced by Ben-Tal et al. [1994], the pq-formulation (which is just the q-formulation with some redundant constraints added) by Quesada and Grossman [1995] and Tawarmalani and Sahinidis [2002], and the *hybrid*-formulation by Audet et al. [2004].

In Haverly [1978] the problem described is for combining oils. For other examples of the pooling problem applied to oil refineries see e.g. Simon and Azma [1983], Baker and Lasdon [1985], Ben-Tal et al. [1994] or Amos and Ronnqvist [1997]. In pooling problems from the oil industry the total mass of each product is typically a variable, whose value is to be determined by the solver. This can cause highly nonconvex problems with an infinite number of local minima, which can vary greatly in objective value. Because of this much of the recent work in pooling problems has been on finding the global minimum. Although the pooling problem originated in the oil industry, like us both Grothey et al. [1999] and Stokes and Tozer [2006] solve pooling problems from the food industry.

	formulation	mixes	nutrients	global
Haverly [1978]	p	single	single	
Lasdon and Waren [1980]	p	single	single	
Baker and Lasdon [1985]	p	single	single	
Floudas and Aggrawal [1990]	p	single	single	
Foulds et al. [1992]	p	multiple	multiple	y
Fieldhouse [1993]	p	single	single	
Visweswaran and Floudas [1993]	p	multiple	multiple	y
Ben-Tal et al. [1994]	q	multiple	multiple	y
Quesada and Grossman [1995]	pq	multiple	single	y
Amos and Ronnqvist [1997]	p	multiple	single	
Adhya and Tawarmalani [1999]	p	multiple	multiple	y
Grothey et al. [1999]	q	multiple	multiple	
Tawarmalani and Sahinidis [2002]	pq	multiple	multiple	y
Audet et al. [2004]	*hybrid*	multiple	multiple	y
Meyer and Floudas [2006]	*hybrid*	multiple	multiple	y
Liberti and Pantelides [2006]	p	multiple	multiple	y
Stokes and Tozer [2006]	q	multiple	multiple	

Table 4.1: Summary of published work on the pooling problem

Table 4.1 is a summary of published work on the pooling problem. This is based on Table 2.1 and Table 2.2 in Almutairi [2008]. The second column of the table shows what formulation is used. The third column shows whether or not the problems considered have a single mix,

like the problem in Figure 4.3, or multiple mixes. The fourth column shows whether or not the problems considered have a single nutrient, like the problem in Figure 4.3, or multiple nutrients. The final column shows whether or not any solution methods detailed are designed to find the global minimum to the problem.

For purpose of comparison with the work in this table we mention that the pooling problems we solve in this thesis are all modelled with what we call the extended q-formulation, and have multiple mixes and multiple nutrients. Most of our solve methods seek only local minima, though some introduced later in Section 6.3 aim to find global minimum.

4.2 Generalised Pooling Problem

A generalised pooling problem (GPP) is a pooling problem where flow is possible from mix to mix. We call pooling problems without this feature standard pooling problems. Although it is suspected that many real life problems are GPPs, almost all of the literature focuses on standard pooling problems. GPPs are however described in Audet et al. [2004] and Meyer and Floudas [2006].

We now describe the two main formulations to model pooling problems, the p-formulation and q-formulation, and introduce the new extended q-formulation. The formulations differ in the extent to which proportional flow variables are used to measure flow in the network, compared to total flow. Figure 4.4 illustrates the difference schematically.

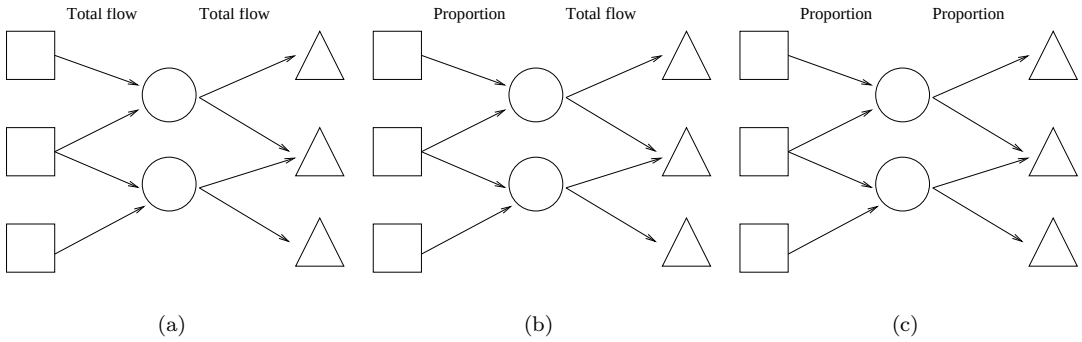


Figure 4.4: Summary of three formulations for the pooling problem; p-formulation (a), q-formulation (b), extended q-formulation (c).

4.2.1 p-formulation

The p-formulation is a natural way to model pooling problems. It is also called the *total flow model*. It originated in Haverly [1978]. In the p-formulation all flow volumes are explicitly measured, with variables for total flow between any two bins. Although the p-formulation is intended for standard pooling problems it could be easily adapted for generalised pooling

problems.

4.2.2 q-formulation

A different formulation for the standard pooling problem was introduced by Ben-Tal et al. [1994]. This is called the q-formulation. In this formulation the variables measuring flow of mass from raws to mixes are replaced with flow proportion variables, that measure the proportion of a mix's input that comes from each raw. Although the q-formulation is also intended for standard pooling problems it could also be easily adapted for generalised pooling problems.

4.2.3 Extended q-formulation

In the extended q-formulation all the variables measuring total flow of raws or nutrients between bins are replaced with flow proportion variables. This formulation was first seen in Grothey et al. [1999], for standard pooling problems. Here we extend it in the natural way to generalised pooling problems, with flow between mixes also measured with flow proportions. The model presented below also has a few other minor features; we have chosen not to eliminate any variables, and have modelled with only equality constraints (with inequalities imposed by simple bounds on variables).

The table above shows the parameters used in defining a problem network.

R, M, P, N	number of raws, mixes, products and nutrients
X_{rn}	nutrient n concentration in raw r
y_{mn}	nutrient n concentration in mix m
z_{pn}	nutrient n concentration in product p
C_r	raw r cost per unit
t_r	mass of raw t
u_m	mass of mix m
V_p	mass of product p to be made
α_{mr}	proportion of mix m input direct from raw r
$\beta_{m'm}$	proportion of mix m' input direct from mix m
γ_{pr}	proportion of product p input direct from raw r
δ_{pm}	proportion of product p input direct from mix m

Table 4.2: Structural parameters and other parameters (all upper case) and variables (lower case) of extended q-formulation

Table 4.2 shows the parameters and variables used in the model. The parameters R, M, P and N that give the dimensions of a model we call the *structural parameters*. Note that only Greek letters are used for flow proportion variables. In the description of the variables, and in the constraints below, we assume that there is a fully interconnected network, where flow is possible between any bins. If flow is not possible between raw r and mix m for example, we can use the simple bounds $\underline{\alpha}_{mr} \leq \alpha_{mr} \leq \overline{\alpha}_{mr}$ to stop this flow. In practice when modelling problems we form various sets to indicate where flow is possible in the network, and only define

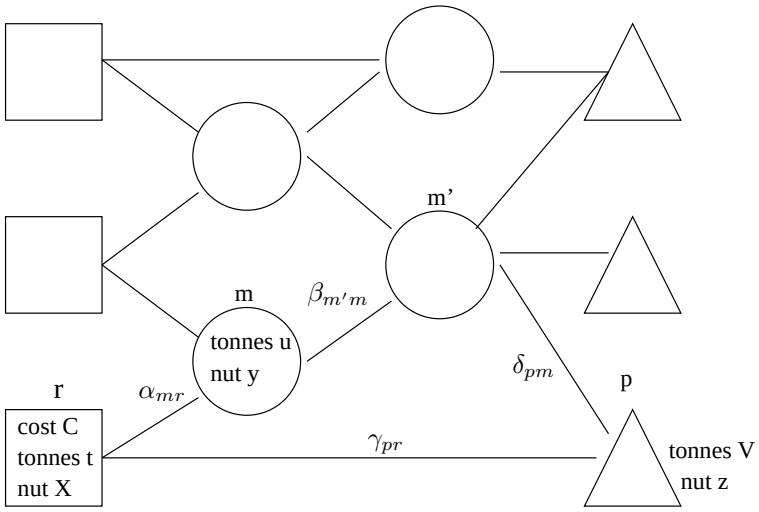


Figure 4.5: Factory3 network

variables and constraints on those sets. But here when presenting the model we prefer not to involve these sets (though they do appear later in equation (4.12).

Figure 4.5 shows an example problem labelled with the parameters and variables. Recall that upper case letters are used for parameters. We represent raws as squares, mixes as circles and products as triangles. Above each bin is the letter used to index it; r for raws, m for mixes and p for products. Each raw has parameters for cost per unit and for nutrient concentrations and a variable for mass. Each mix has variables for mass and for nutrient concentrations. Each product has a parameter for mass and variables for nutrient concentrations. The flow proportion variables measure the input proportion that a bin gets from the bins that precede it in the network. We therefore refer to this model as *backward measuring*. Apart from in exceptional circumstances (such as *drying*, see later), the flow proportion variables take values between zero and one.

$$\min \sum_{r=1}^{R} C_r t_r \tag{4.5}$$

$$\text{s.t.} \sum_{r=1}^{R} \alpha_{mr} X_{rn} + \sum_{m=1}^{M} \beta_{m'm} y_{mn} = y_{mn} \qquad m' \in 1 \dots M,\ n \in 1 \dots N, \tag{4.6}$$

$$\sum_{r=1}^{R} \alpha_{m'r} + \sum_{m=1}^{M} \beta_{m'm} = 1 \qquad m' \in 1 \dots M, \tag{4.7}$$

$$\sum_{r=1}^{R} \gamma_{pr} X_{rn} + \sum_{m=1}^{M} \delta_{pm} y_{mn} = z_{pn} \qquad p \in 1 \dots P,\ n \in 1 \dots N, \tag{4.8}$$

$$\sum_{r=1}^{R} \gamma_{pr} + \sum_{m=1}^{M} \delta_{pm} = 1 \qquad p \in 1 \dots P, \tag{4.9}$$

$$\sum_{m=1}^{R} \alpha_{mr} u_m + \sum_{p=1}^{R} \gamma_{pr} V_p = t_r \qquad r \in 1 \dots R, \tag{4.10}$$

$$\sum_{m'=1}^{M} \beta_{m'm} u_m + \sum_{p=1}^{M} \delta_{pm} V_p = u_m \qquad m \in 1 \dots M. \tag{4.11}$$

The main mathematical model of this formulation is given above. Note that these are all equality constraints. Any inequalities are then expressed as simple bounds on these variables (not shown). For example, a nutrient concentration bound on a product is contained in simple bounds on $\boldsymbol{z}$.

In the objective of (4.5) the total cost is minimised, by minimising the sum of the cost of all the raws. Constraint (4.6) defines the nutrient concentration in each mix, by considering the flow into that mix from raws and the flow from other mixes. Constraint (4.7) says that the sum of proportional flows into each mix is one. Constraint (4.8) defines the nutrient concentration in each product, by considering the flow into a product from raws and from mixes. Constraint (4.9) says that the sum of proportional flows into each product is one. Constraint (4.10) defines the mass of each raw, by considering the flow of that raw into mixes and products. Constraint (4.11) defines the mass of each mix, by considering the flow of that mix into other mixes and into products.

Constraints (4.6), (4.8), (4.10) and (4.11) are all potentially nonlinear. These constraints can all contain terms with a flow proportion variable multiplied by either a nutrient concentration or mass variable. Where one variable is multiplied by another this is called a *bilinear term*. Bilinear terms are the only type of nonlinear term in the model of the extended q-formulation,

We call a variable nonlinear if it may appear in a nonlinear term, otherwise we call it linear. The nonlinear variables of the extended q-formulation are therefore $\boldsymbol{u}$ and $\boldsymbol{y}$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\delta}$. By considering Figure 4.5 again we note that all the nonlinear variables are involved with the mixing bins. Typically not many of the variables of a pooling problems are nonlinear. Figure 4.6 shows the location of each of the variables in the constraint matrix of a linearisation of a GPP called

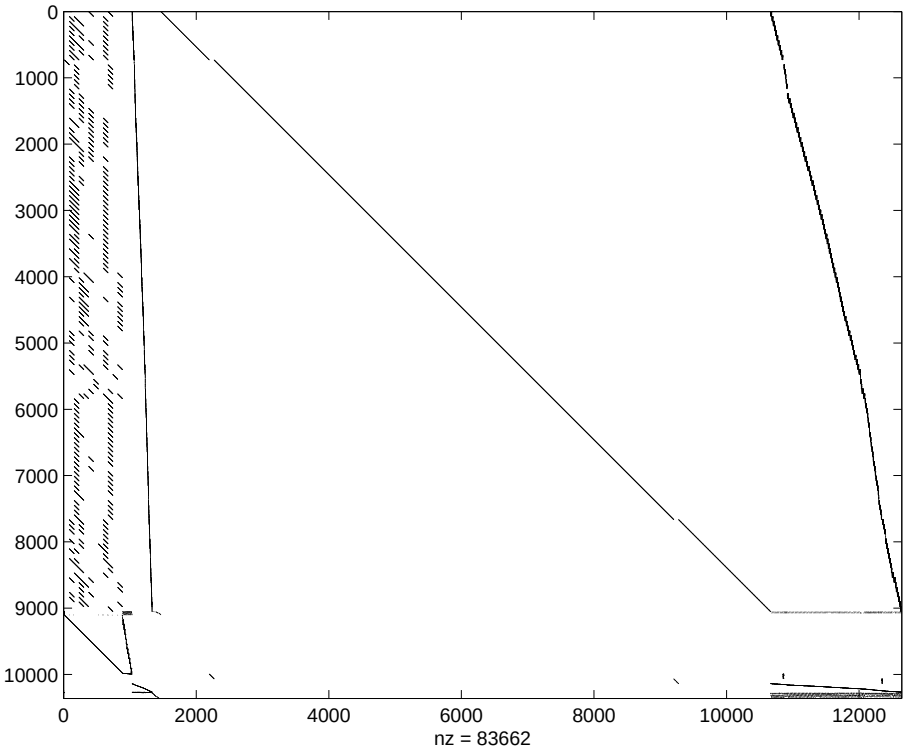


Figure 4.6: Sloten1relax linearised constraint matrix

Sloten1relax. From left to right the columns represent $\boldsymbol{u}$ (12 columns), $\boldsymbol{y}$ (876), $\boldsymbol{\alpha}$ (144), $\boldsymbol{\delta}$ (302), $\boldsymbol{t}$ (132), $\boldsymbol{z}$ (9198) and $\boldsymbol{\gamma}$ (1978). The majority of the variables in the problem are of the type $\boldsymbol{z}$, and overall nearly 90% of the variables are linear. For this problem a large proportion of the constraints are of type (4.8), where the row defines a $\boldsymbol{z}$ variable which appears nowhere else in the model. Such rows could be reformulated by eliminating $\boldsymbol{z}$. Eliminating variables is considered in Section 5.2.1.

In the extended q-formulation there are nonlinear constraints (4.6) and (4.8), for the nutrient concentrations in mixes and products respectively, and nonlinear constraints (4.10) and (4.11), for the masses of raws and mixes respectively. Let the sets S^α, S^β and S^γ contain all the nontrivial variables of type α, β and γ respectively. Then totalling the number of nonlinear terms in the extended q-formulation gives:

$$N|S^\beta| + N|S^\delta| + |S^\alpha| + |S^\beta|, \tag{4.12}$$

where each of the four expressions in the sum corresponds to constraints (4.6), (4.8), (4.10) and (4.11) respectively.

We define a *splitting network* as one where few raws supply many products, so the raws are split between the products. In a *pure splitting network* there is always only one input to each bin. We define a *collecting network* as one where many raws are collected into few products.

An example of a collecting network is H_prob, shown later in Figure 4.8. In a *pure collecting network* every bin always has only one output.

As it is backward measuring the extended q-formulation is good for modelling splitting networks. In particular:

Theorem 3. *For a pure splitting network, a backward measuring formulation like the extended q-formulation is entirely linear.*

As each mix and product in a pure splitting network has only one input all of the flow proportions variables must be equal to one. Hence the flow proportion variables are all constants, and all the bilinear terms in the model become linear. This proves the theorem. Conversely it can be shown that a forward measuring formulation is linear for pure collecting networks. This perhaps explains why the p-formulation, which is forward measuring, is popular for problems from the oil industry, which can be close to pure collecting networks.

As well as the extended q-formulation we also modelled some GPP test problems with the p-formulation. In this limited comparison the extended q-formulation was found to have far fewer bilinear terms than the p-formulation, and was much better at finding feasible solutions when solving with SLP. The extended q-formulation is used for the rest of the thesis.

4.2.4 Drying

A novel feature of some of our test problems is that *drying* may occur. In a problem that allows drying some mixing bins do not have the normal role of mixing ingredients, but instead dry the mix they are given, by reducing the amount of the water raw material. The possibility of drying is represented in the model by allowing a negative flow proportion of the water raw material into some bins, and allowing a maximum flow proportion of greater than one for the other inputs.

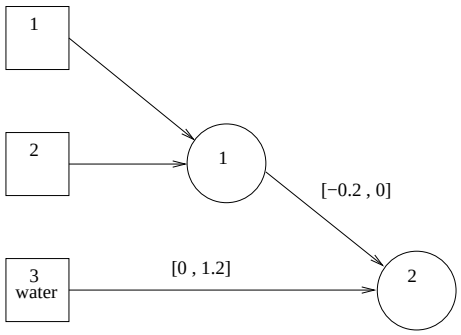


Figure 4.7: Network with possible drying

Figure 4.7 shows part of a GPP where drying can occur. We use the notation $[l, u]$ for lower and upper bounds on a variable, here the flow proportion variables for input into Mix 2. Raw 1 and Raw 2 combine to form Mix 1. To represent the fact that Mix 1 can be dried the dummy Mix 2 has been included in the network. Mix 2 has its only input Mix 1 and Raw 3 (water). If drying

	R	M	P	N	drying possible	multiple local minima
Factory3	3	4	3	2	y	
H_prob	39	21	1	20	y	
Sonoco97	32	6	25	8		y
Global_data	69	86	16	48	y	
G1b	69	8	2	48	y	
Plant1a	55	62	10	43	y	
Sloten1relax	132	12	126	73	y	y
Sept98	33	7	27	11		y
Jan99	32	6	32	14		y
Gns	58	7	18	17		y
Nville	62	7	40	17		y
Hen040304	177	287	134	65	y	

Table 4.3: Dimensions and features of GPP test problems

occurs then there is a negative flow proportion to Mix 2 from Raw 3, and a flow proportion to Mix 2 from Mix 1 of greater than one. Mix 2 would then contain a more concentrated version of Mix 1. The bounds in this example reflect the maximum observed proportion of drying allowed in any of the test problems, of 20%.

4.2.5 GPP test problems

We now present our GPP test problems. Apart from the tiny test problem Factory3 all these problems are based on real factories, and are supplied by Format International. The problems come from different Format clients, for example Sonoco97 is a problem from 1997 supplied by the global packaging company Sonoco. The real problems typically have several dozen bins. The largest, Hen040304 has several hundred raws, mixes and products. Using our extended-p formulation this problem has over 30,000 variables and over 30,000 constraints. Details such as the actual size of each factory, how frequently the factories are optimised, and the true costs involved are not known to us.

Table 4.3 gives the number of raws, mixes, products and nutrients, as well as whether or not each problem allows drying and if we have ever found multiple local minima. Although the problems are all nonlinear, several problems, especially the smaller ones, appear to have only one local minimum. We call the first seven problems test set GPP1. The six problems with multiple local minima, and the large test problem Hen040304, we call GPP2.

All problems will be modelled and solved with the standard model, which is the extended q-formulation. In these test problems each mass variable t_r usually has a lower bound of zero and may have an upper bound. Each mass variable u_m is usually unbounded. The nutrient concentration and flow proportion variables have lower and upper bounds, for example the mix nutrient concentration of mix m and nutrient n, y_{mn}, is bounded by $\underline{y}_{mn} \leq y_{mn} \leq \overline{y}_{mn}$. A few of the problems have specific extra conditions, which need extra constraints to model them. For example, a condition may exist that one product must have more of Raw 1 than Raw 2.

Although a condition such as this is linear, to model it might need extra nonlinear constraints, to track the flow of Raw 1 and Raw 2 through the network.

Factory3

Factory3 is a very small problem whose network was shown in Figure 4.5.

H_prob

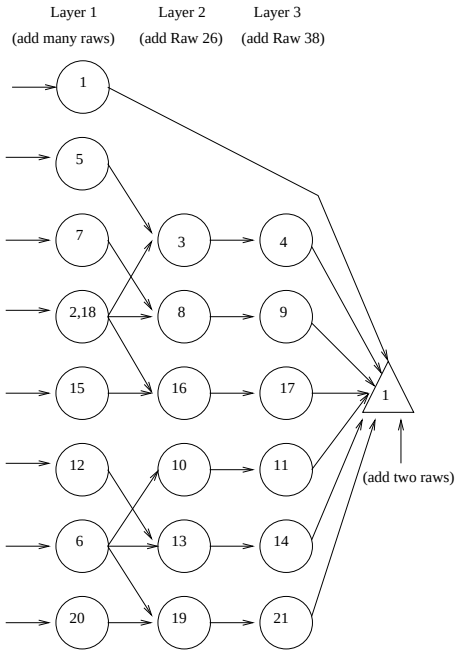


Figure 4.8: H_prob

H_prob is a small problem with tightly constrained flow between bins. It is a collecting network, where many raws are collected into few (in this case one) products. Figure 4.8 shows the network, omitting the raws. There are three layers of mixes flowing into the single product. Most of the raws are added at Layer 1. One raw is added at Layer 2. Layer 3 is for drying, where potentially negative amounts of Raw 38 (water) may be added. Two raws are added directly to the product.

Sonoco97

Figure 4.9 shows a schematic of the network of Sonoco97. This is a standard pooling problem (as opposed to a generalised pooling problem), with no flow possible from mix to mix. There are sixteen distinct raws, but these are duplicated to make two sets. The first set reaches the products via a bottleneck of six mixes. The second set are straights that flow directly into the products, at double the usual price. The presence of straights guarantees that the problem is always feasible. The products have lower and upper bounds on nutrient concentrations of $0.9\boldsymbol{z}^*$ and $1.1\boldsymbol{z}^*$ respectively, where $\boldsymbol{z}^*$ is a given target value. No drying is possible.

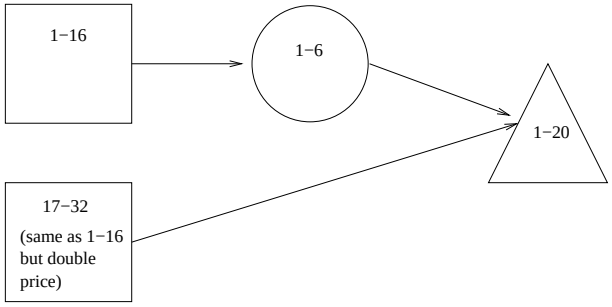


Figure 4.9: Sonoco97

Each arc from raws to mixes and from mixes to products has a lower bound on flow proportion of zero, and an upper bound of one. There are also no bounds on mix composition. We call a problem with these properties, implying unbounded flow in and out of mixes, *free flowing*. Free flowing problems, such as Sonoco97, are typically highly nonlinear. Furthermore, in Sonoco97 the duplication of the raws and presence of many identical mixes means many local minima exist. The nonlinearity of Sonoco97 is considered again in Section 6.3.

Global_data

Global_data is a large problem with many mixes. Figure 4.10 shows the network, omitting the raws. At Layer 1 most of the raws are added to the network. A few are added at later layers. At Layer 2 the mixes combine. At Layer 3 a few more raws are added. Layer 4 is for drying. A few raws go directly into the products. Several pairs of products have identical inputs, hence they are shown in the same position in the network. We call these *product groups*. The products in different product groups have no mixes in common, although they may share raws. As there are no upper bounds on raw mass the problem is in fact decomposable by product group. Subsections of this factory can therefore be treated and solved as if they were separate factories, which is much more efficient than solving them together. If raw mass bounds are added the problem no longer decomposes.

In Chapter 6 a multifactory problem, G, is formed, using subfactories of Global_data. The component factories of G are the first three product groups of Global_data, each with their associated mixes and raws. A transport problem links the factories and imposes upper bounds on total raw mass used between them.

G1b

The first product group of the decomposition of Global_data consists of Product 1 and Product 14. The subfactory consisting of just the first product group, and the associated mixes and raws, is called G1b. G1b has the same number of raws and nutrients as Global_data, but far fewer mixes and products.

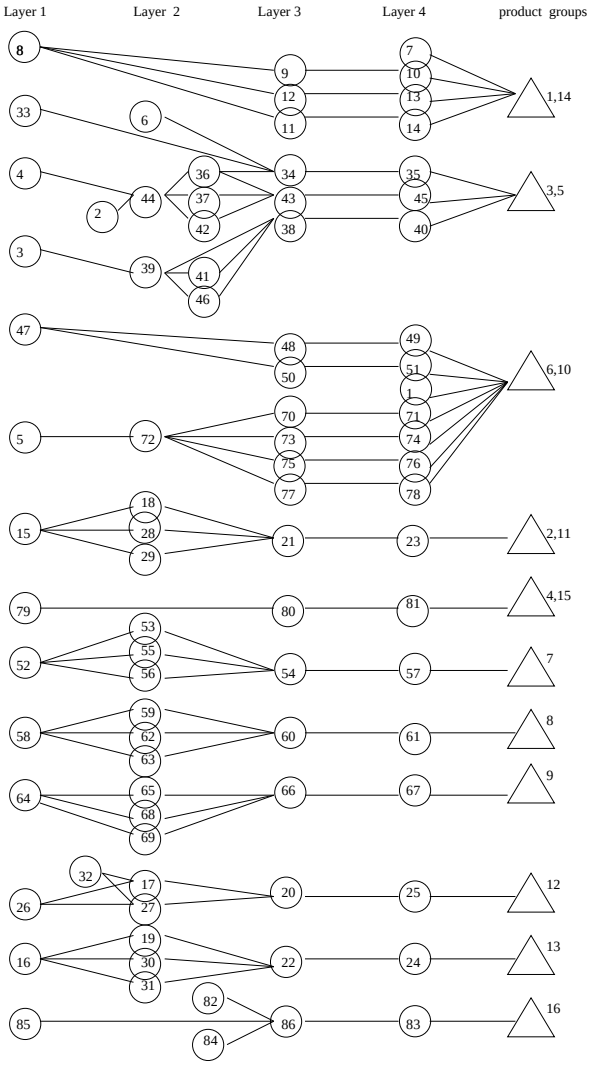


Figure 4.10: Global_data

Plant1a

Plant1a is a large problem that is similar to Global_data. It is also possible to decompose Plant1a into distinct product groups. In Chapter 6 a multifactory problem, R, is formed by combining twelve different versions of Plant1a, each with different raw costs, and adding a transport problem imposing upper bounds on total raw mass used between the factories.

Sloten1relax

Sloten1relax is a large diet problem with many bins. It is the continuous relaxation of a diet problem called Sloten1, which has integer constraints. The original Sloten1 problem is introduced and solved in Chapter 7.

Sept98,Jan99,Gns,Nville

Sept98, Jan99, Gns and Nville are all standard pooling problems with a similar network to Sonoco97. They all have a huge number of local minima, which makes the global minimum of these problems hard to find.

Hen040304

Hen040304 is a complicated problem with a network containing many layers of mixes. The model has over 30,000 variables and over 30,000 constraints. It is the largest diet problem that we solve.

4.3 Role of the mixes

In this section we look in more detail at the effect of having mixing bins in the model. Although it is well known that it is the presence of the mixing bins that makes GPPs nonlinear, in this section we explore the role of the mixing bins more closely. In Section 4.3.1 (and partly in Section 4.3.2) we see that the mixing bins can have a large effect on the objective value, in Section 4.3.2 we see that they can lead to infeasible problems, and in Section 4.3.3 we see how they create local minima.

4.3.1 Mix relaxations

We first consider what we call *mix relaxations*. Mix relaxations are relaxations of bounds on the variables that are involved with the mixes. These are the variables for mix nutrient concentration ($\boldsymbol{y}$), mix mass ($\boldsymbol{u}$) and flow proportion in or out of mixes ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\delta}$). Note that these are all the nonlinear variables. These relaxations are different to those in the literature, as they are for the new extended q-formulation, and are applied to a generalised pooling problem with flow from mixing bin to mixing bin.

All the mix relaxations we consider result in problems which are still nonlinear. Indeed, all the mix relaxations we consider are harder to solve than the original problems. Four different mix relaxations that are applied to GPP are described below. We use the notation $[l, u]$ for lower and upper bounds on a variable.

$rm1$ All mix nutrient concentration bounds relaxed to $[0, \infty]$.

$rm2$ Bounds on flow proportion from raws to mixes and from mixes to products relaxed to $[0, 1]$ wherever bounds were not previously $[0, 0]$.

$rm3$ Relaxations $rm1$ and $rm2$ combined.

$rm4$ Relaxations $rm1$ and $rm2$ combined, and furthermore all mix nutrient concentration bounds relaxed to $[0, \infty]$, and all bounds on flow proportion to mixes from rows and to products from mixes relaxed to $[0, 1]$.

In practice if drying can occur flow proportion bounds of wider than $[0, 1]$ are used. The four mix relaxations get progressively tighter from $rm1$ to $rm4$, apart from $rm1$ and $rm2$, where neither relaxation is tighter than the other for all problems. $rm3$ and $rm4$ are similar, but in $rm3$ only some raws can flow into mixes, and only some mixes can flow into products, whereas in $rm4$ there is unlimited flow from raws to mixes and mixes to products. In fact as it removes any nutrient concentration bounds on the mixes $rm4$ results in a free flowing problem.

	Factory3	H_prob	Sonoco97	Global_data	G1b	Plant1a
$rm1$	0.982	0.952	1.000	0.995	1.000	1.000
$rm2$	0.975	0.711	1.000	0.873	0.992	0.929
$rm3$	0.958	0.697	1.000	0.865	0.991	0.928
$rm4$	0.942	0.697	1.000	fail	fail	fail

Table 4.4: Ratio of mix relaxation best objective value to standard model best objective value

Table 4.4 compares the objective value of the mix relaxations to the objective value of the standard model. We solve with an sequential linear programming (SLP) code that we call Slp5. The purpose is to discover how much the presence of the mixes, and their associated constraints, is limiting the objective value. For the results in this table we are therefore only interested in the objective value and not the speed of the SLP solve (the speed of different solvers is considered in the next chapter). For each mix relaxation we solve at least three times and use the best objective value of the three solves. A small ratio in the table indicates the mix relaxation had a much lower objective value than the standard model solve. Some problems failed to solve with $rm4$. For Sonoco97 the standard model is already free flowing, so all mix relaxations are equivalent to the standard model (though different local minimum may be found on different solves).

On all problems that could be solved $rm2$ has an equal or lower objective value than $rm1$. This shows that relaxing the flow proportion bounds has a more significant impact than relaxing the mix nutrient concentration bounds. This may simply be because for these test problems the flow proportion bounds were added by the problem modellers to implicitly constrain the nutrient concentrations.

Where $rm4$ solved there was only a small difference between the $rm3$ and $rm4$ solutions, indicating that the objective value is not greatly constrained by only allowing few raws to flow into mixes and few mixes to flow into products. This may be because the problem modellers placed bounds of zero on flows that were anticipated to be zero.

For the Global_data and especially H_prob problems the mix relaxation objective values are significantly lower than the standard model objective values. This could indicate that the bounds in the standard model imposed on variables involved with the mixes are highly detrimental to the profitability of the factory. However, in practice these bounds may have been used to implicitly model conditions on the products, as is known to be the case for Sloten1relax. In this case the mix relaxations are in fact implicit product relaxations.

4.3.2 Composition of mixes

Grothey et al. [1999] attempted to solve standard pooling problems (those with no mix to mix flow) using the observation that fixing the mix compositions makes a problem linear, and in fact decomposes it into M LPs to form the mixes and P LPs to form the products. Fixing the mix compositions also makes a GPP linear. Solving a GPP can therefore be considered to be the problem of what composition to make the mixes, as once the mixes are fixed the problem is an LP and so is easy to solve. Although most of what follows applies equally to standard and generalised pooling problems, for simplicity we consider only standard pooling problems for the remainder of this chapter, and furthermore assume no drying is possible.

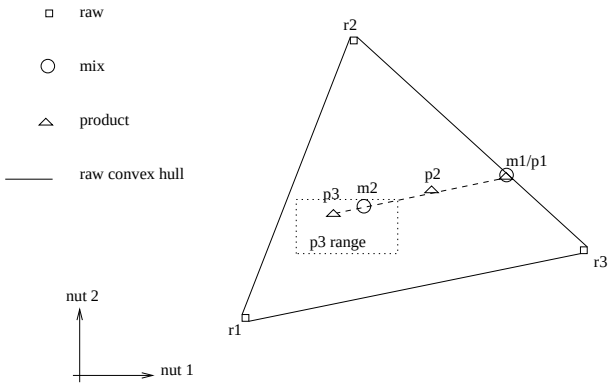


Figure 4.11: Nutrient space of 3-2-3 problem

Fixing the mix compositions can be thought of as placing each mixing bin in a multi-dimensional space spanned by the nutrients. This new approach, based on the observations of Grothey et al. [1999], is somewhat similar to the graphical view a problem included in Haverly [1978], where he considers where to place the single mix in the multi-dimensional space spanned by proportions of input from the different raws. Figure 4.11 shows the nutrient space for a small example problem, with two nutrients spanning the horizontal and vertical axes. We call this a 3-2-3 problem, meaning there are three raws, two mixes and three products. In the diagram we

abbreviate names, with e.g. $m1$ standing for Mix 1.

It is useful to introduce some terminology at this stage. A bin that has identical lower and upper bounds for each nutrient concentration is called *fixed*, and can be represented by a single point in the nutrient space (e.g. $r1$ or $p1$ in Figure 4.11). An unfixed bin is represented by an appropriately dimensioned rectangle (e.g. $p3$ in Figure 4.11). In pooling problems all raws are fixed, and mixes and products are either fixed or unfixed.

The *raw convex hull* is the smallest convex shape in the nutrient space that contains all the raws. In Figure 4.11 there are three raws, $r1$, $r2$ and $r3$, spanning the raw convex hull, which contains all the mixes and products. In a feasible solution all mixes and products must lie inside the raw convex hull. For any given composition of each product the *product convex hull* is the smallest convex shape in the nutrient space that contains all the products. If some products are not fixed there are several possible product convex hulls. In Figure 4.11 the product convex hull depends on the position of Product 3.

For a problem with N nutrients we denote the dimension of the space spanned by the nutrients as N'. This may be less than N, if some nutrients are redundant so do not add to the dimensionality. This can happen, for example, if all raws have equal concentration of a nutrient, or if all mixes and products have ranges for a nutrient which contain all possible nutrient concentrations. The dimension of the smallest space that spans at least one of the product convex hulls is denoted N'^P. Where there are many products it is usual for N'^P to be equal to N', else it may be less. In Figure 4.11 neither of the two nutrients is redundant, hence $N' = N = 2$.

We refer to bins with compositions near the centre of the nutrient space as *central*. A bin whose output flows into another bin is called an *input bin*, hence bins that flow into products are called *product input bins*, or simply *product inputs*.

Observation 3. *For a feasible solution the number of product input bins should normally be at least* $N'^P + 1$.

We say *normally* as, in exceptional circumstances, for example if a product can be made directly out of one raw, fewer product inputs will suffice. However, usually there must be enough product inputs to span the dimensionality of the products in the nutrient space. With this observation in mind we now reconsider Figure 4.11. As there are no straights the only product inputs are the mixes, $m1$ and $m2$. By Observation 3 if the problem is to be feasible the dimensionality of the products, N'^P, must be only one. This implies that the three products, $p1$, $p2$ and $p3$, must be degenerate in the nutrient space, so the unfixed product $p3$ must be placed in a line with the fixed products $p1$ and $p2$. Such a placement of $p3$ is shown in the diagram.

We now consider where to place the mixes in the nutrient space to give the solution with lowest objective value. For simplicity we further restrict ourselves to only considering free flowing problems (those where flow from raws to mixes and mixes to products is unconstrained,

and there are also no bounds on mix composition) which have no straights and no raw mass upper bounds. First some cases are trivial:

Theorem 4. *In a free flowing problem if there are as many mixes as raws, or as mixes as products, the problem reduces to the blending problem.*

If there are enough mixing bins then the problem is equivalent to directly making each product out of the raws, as in the blending problem. Specifically if there are at least as many mixes as raws it is clear that placing one mix directly over (i.e. at the same composition as) each raw is optimal, and so the problem reduces to the blending problem. Similarly if there are at least as many mixes as products placing one mix directly over each product is optimal, and again the problem reduces to the blending problem. This proves the theorem.

In these simple cases the price of the products is cheap. If there is a limited number of mixing bins the cost of making each product is more expensive. We refer to the *premium* of a product as the difference between its cheapest cost when it is made directly from raws, and its cost in the actual solution. Solving a pooling problem can be viewed as minimising the total of the premiums for all products, weighted by the mass of each product to be made.

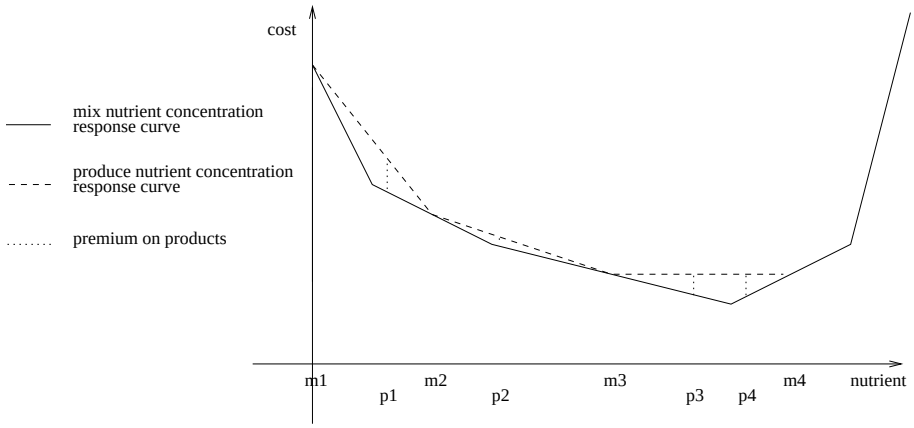


Figure 4.12: 6-4-4 problem showing premiums on products

Figure 4.12 shows the premiums on products for a given mix placement on an example free flowing 6-4-4 problem with one nutrient, where all products are fixed. Each mix is made out of raws. The price of a mix depends on the nutrient concentration of the mix, and is given by the mix nutrient concentration response curve. This curve also gives the minimum price of a product, were the product also to be made directly out of raws. The actual product nutrient concentration response curve is determined by the position of the mixes in the nutrient space, and lies above the mix nutrient concentration response curve in some places. The difference between the two response curves is the premium paid for making a product out of mixes, instead of directly out of raws. In Figure 4.12 the products are all fairly central, so for this problem placing a mix at each of the four turning points of the mix nutrient concentration response curve lets the products be made at zero premium, so is the optimal solution for this

problem. With fewer mixes the optimal product curve is more expensive than the mix curve, but in a good solution will be close to the mix curve at the nutrient concentrations where the products are positioned. By increasing the number of mixes the premiums can be reduced, until eventually there are as many mixes as raws or products, and by Theorem 4 there are no premiums, wherever the products are situated in the nutrient space.

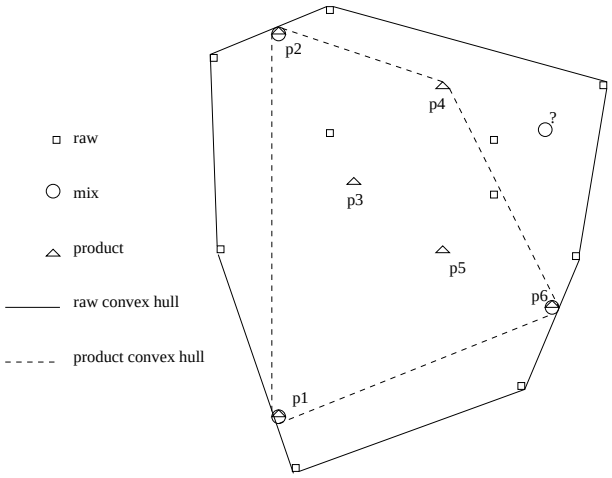


Figure 4.13: Nutrient space of a free flowing 10-4-6 problem

We now consider where to place the mixes to minimise the premium. The mixes should usually be placed to span a small convex hull around the products. However, there are exceptions. In Figure 4.13 the four mixes must be placed to supply the six products. For a feasible solution three of these mixes should go directly over products $p1$, $p2$ and $p6$ (as shown). This leaves the final mix spare, it can occupy a range of possible positions and still give a feasible solution. The optimal placement of the spare mix depends on the raw costs, and for certain costs is outside the product convex hull.

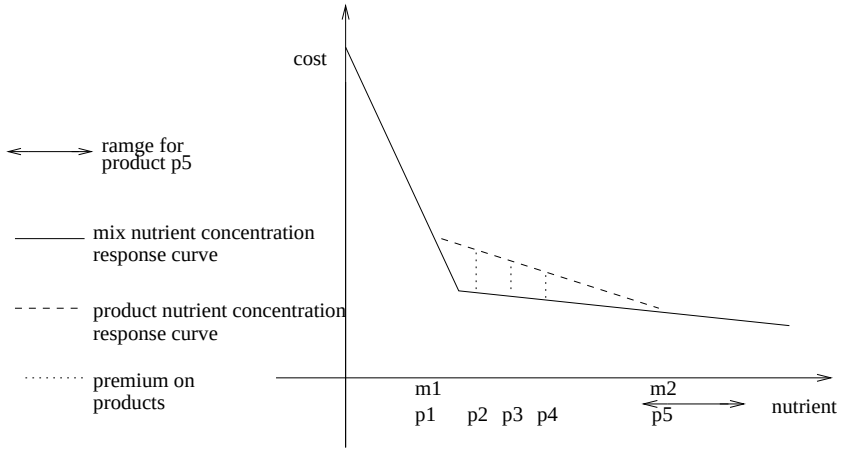


Figure 4.14: Mix nutrient concentration response curve of a free flowing 3-2-5 problem

In the optimal solution unfixed products are usually placed at the position in their nutrient concentration range that allows them to be made most cheaply. As before there are exceptions

though. Figure 4.14 shows the mix nutrient concentration response curve of an example problem. For a feasible solution the two mixes must each be placed directly over one of the outer products, $p1$ and $p5$. $p5$ is unfixed, and at the optimal solution (shown in the diagram) is placed at the expensive end of its range.

When there are fewer mixes than products there are two strategies of placing each mix. The first is to place the mix directly over one of the products. With this strategy some products are made very cheaply, and some are expensive. The second is to place the mix close to several products, but not directly over any of them. With this strategy all products are made reasonably cheaply. We now consider which of these two alternatives is better, by summing the premiums on products costs for each strategy. This is done for two example problems.

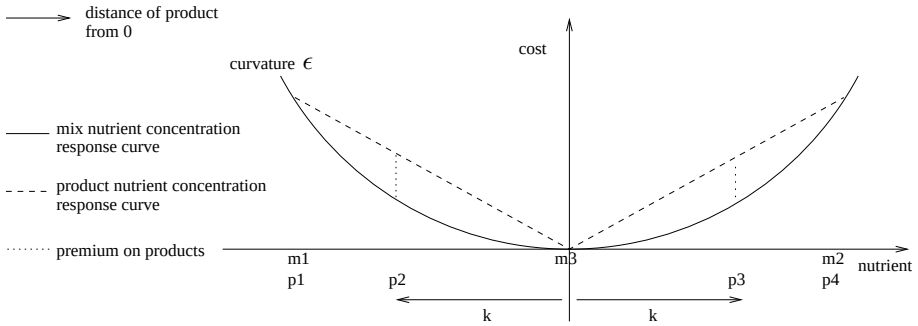


Figure 4.15: Mix nutrient concentration response curve and product nutrient concentration response curve of an R-3-4 problem

Figure 4.15 is one of a family of problems, parametrised by curvature ϵ of the mix nutrient concentration response curve and distance k between the origin and $p2$ or $p3$. The four products all have equal masses. For a feasible solution one mix each is required for the two outer products, $p1$ and $p4$, leaving one mix spare. This spare mix should be placed so that products $p2$ and $p3$ can be made as cheaply as possible. This means either placing the mix directly over $p2$ or $p3$, or placing it centrally between them, at the origin. Which of these two alternatives is better depends on the values of k and ϵ. For this example an analytic solution can be found, that shows when ϵ is small, or when k is small, the better solution is to place the spare mix centrally, which is the solution shown in the diagram.

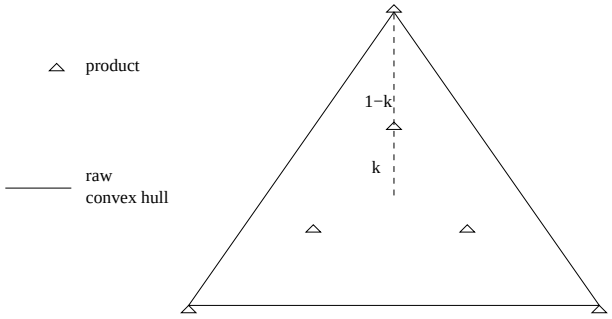


Figure 4.16: Nutrient space of one of a family of 6-4-6 problems

Figure 4.16 is one of a family of two dimensional problems, parametrised by distance k from the origin to the three inner products. The six raws (not shown) are in the same positions as the products, with three cheap inner raws and three expensive outer raws. For a feasible solution three of the four mixes are placed directly over the outer three products, leaving one mix spare. As in the problem above, the question is whether this spare mix should be placed directly over one of the central products, or placed centrally between them. For small k the best solution is again to place the spare mix centrally. This example problem and the one above collectively suggest that if and only if the products are clustered, it is best to place a spare mix centrally between them. This leads to the observation below:

Observation 4. *If products requiring similar nutrient concentrations are being made it is best to place the mixes between them, if product with different nutrient concentrations are being made it is best to make some of them exactly with the mixes.*

4.3.3 Local minima

We saw above that the pooling problems are nonlinear because of the mixing bins. We now look at three new example problems that illustrate how the mixing bins cause local minima. We first consider an example based on the problem shown in Figure 4.16, but with the raw placements perturbed slightly from that problem. This makes the problem asymmetric and so creates local minima with different objective values. The resulting problem is named Ex1. In the solution to Ex1 three of the four mixes are again placed directly over the outer three products, leaving one mix spare. Each of the three inner products is made from the spare mix and two of the outer mixes, which two depending on where the spare mix is placed. The placement of the spare mix therefore affects the cost of the products, as well as affecting its own cost.

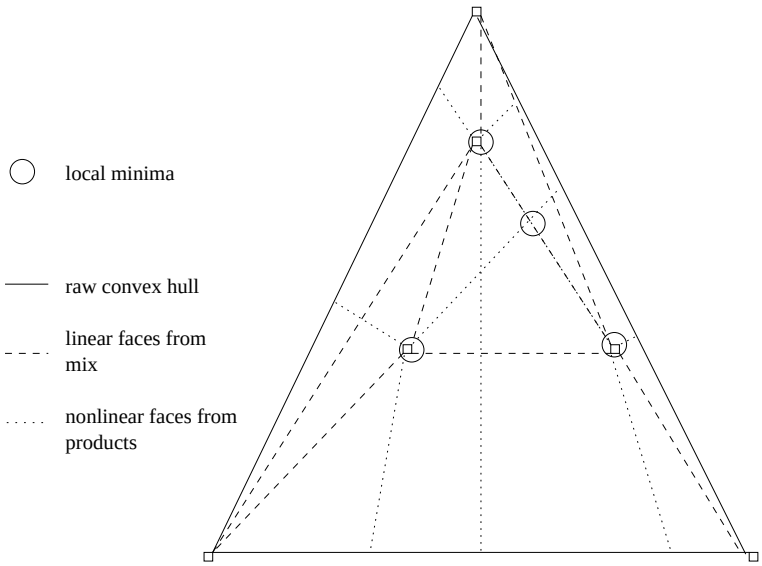


Figure 4.17: Overhead view of mix nutrient concentration response surface for spare mix of Ex1

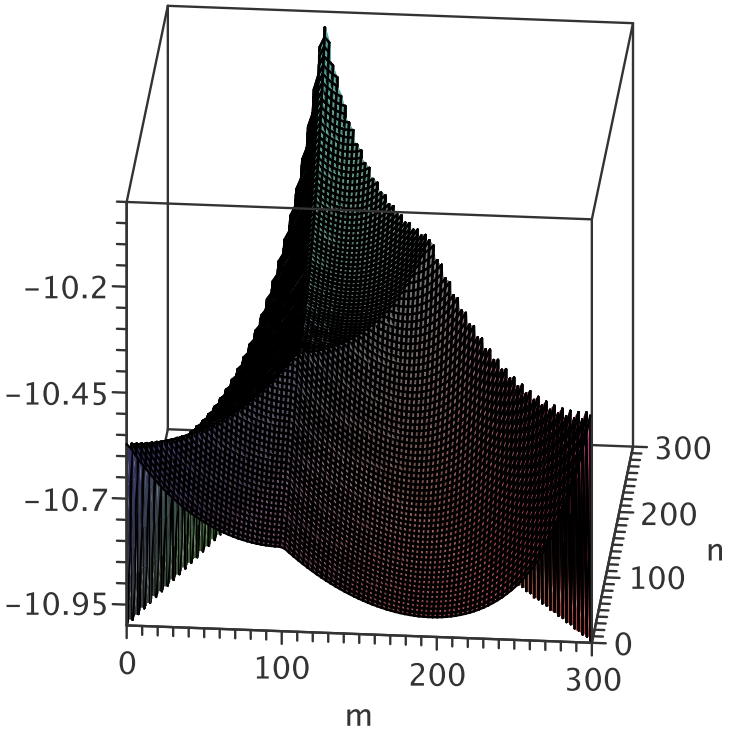


Figure 4.18: Central triangle of mix nutrient concentration response surface for final mix of Ex1, inverted

The 2D surface of the mix nutrient concentration response curve for the spare mix is made from the combination of nonlinear faces which come about from the placement of the spare mix affecting the cost of the products, and linear faces which come about from the placement of the spare mix affecting its own cost. Figure 4.17 shows a plan view of this surface. The four local minima of this problem are contained within a central triangle, that consists of four faces, all of which are concave. Three of the local minima are at the corners of the central triangle, which corresponds to placing the spare mix directly over one of the three inner products. This central triangle is shown in more detail in Figure 4.18. For display purposes the graph is inverted so that the four local minima now stand out as maxima. Also for display purposes outside the central triangle the surface is shown as dropping away steeply.

Figure 4.19 shows the nutrient space of a second example problem. There are six products in a central ring, which must all be spanned by the three mixes. For this problem $N'^P = 2$, as the products are all in a 2D plane. Hence by Observation 3 for a feasible solution all three mixes are needed. There are in fact infinitely many feasible solutions, the cost of each one depending on the cost of placing mixes in different areas of the nutrient space, which is determined by the raw costs. Example solution 1 is an example of what is expected to be a cheap solution, as the

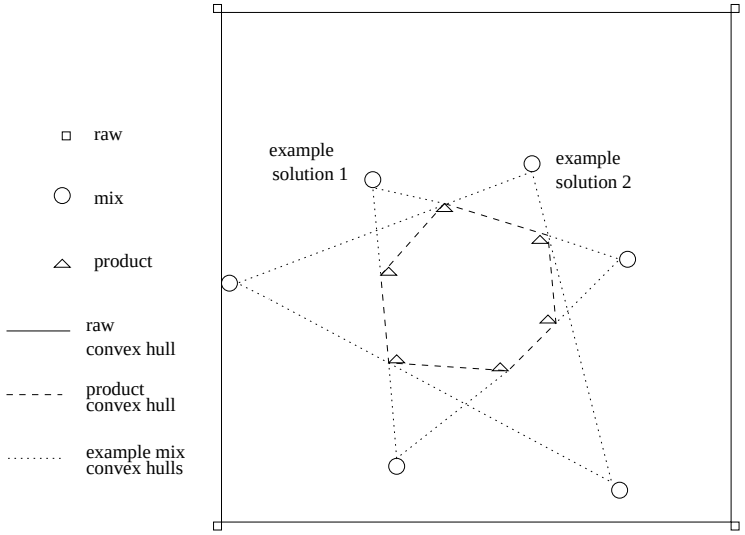


Figure 4.19: Nutrient space of an R-3-6 problem. Two possible feasible mix placements are shown

mixes fit tightly around the products. This means the mixes are central so can be made out of central raws, which are expected to be cheap. Example solution 2 is expected to be more expensive as the mixes do not fit tightly around the products. For a feasible solution all three of three mixes must be placed outside the product convex hull. This leads to a highly nonconvex feasible region for each mix. Extra mixes, or straights, would make the problem easier to solve.

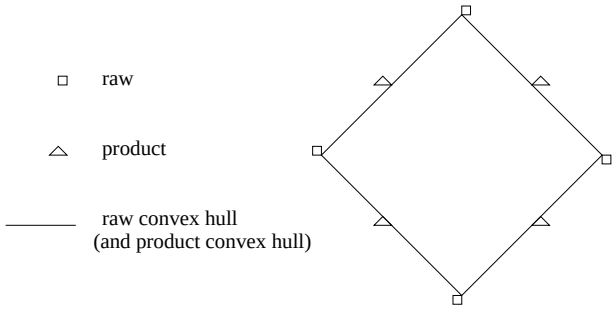


Figure 4.20: Nutrient space of a 4-4-4 problem

Figure 4.20 shows the nutrient space of a third example problem. This problem has extreme nonconvexity. There are four products to be made and four mixes available. All four products are on the edge of the raw convex hull, and consequently also on the edge of the mix convex hull. There are therefore two very different solutions; either the four mixes are placed directly over the four products or alternatively the four mixes are placed directly over the four raws. Any intermediate placement of the mixes, such as a convex combination of the two feasible solutions, is infeasible. This problem demonstrates the extent of nonconvexity that is possible with pooling problems.

The decomposition method of Grothey et al. [1999] required fixing the mix compositions, using an approximation of the feasible region for each mix. The method struggled, because of

the nonconvexity of these regions.

4.4 Summary

In this chapter we have described the GPP as an extension of the blending problem. The GPP will be the basis for the problems solved in the following three chapters. A new formulation, the extended q-formulation, is used to model the problem.

Nonlinearity in the GPP is caused by multiple inputs and outputs to the mixing bins. The mixing bins lead to highly nonconvex feasible regions and multiple local minima. By relaxing bounds on the variables that apply to the mixing bins a much better objective value is possible on some problems.

Solving a GPP can be seen as the problem of where to place the mixes in the nutrient space,. This gives insight into where to place the mixes to give a feasible solution, and one with good objective. Minimising the objective can be thought of as minimising the premium of making products out of mixes rather than directly out of raws. We showed how the feasible region for each mix can be highly nonconvex, which explains the difficulties of solving a pooling problem by decomposition.

4.4.1 Further work

In this chapter we have introduced a geometrical insight into solving diet problems. It is possible that this could insight could form the basis of an algorithm, to solve or at least presolve a GPP. It would also be interesting to adapt some of the existing work on standard pooling problems (formulations, relaxations. etc.) to the generalised pooling problem, and compare it to the work in this thesis.

Chapter 5

Solving the Generalised Pooling Problem

In Chapter 4 we introduced a diet problem called the generalised pooling problem (GPP). In this chapter we look at solving that problem.

In Section 5.1 we review existing methods to solve GPPs, including using an SLP solver named Integra_1. We confirm that SLP is effective for solving pooling problems, in particular we show it is effective for GPPs. In Section 5.2 we describe improvements to Integra_1 that go into forming a new solver, named Integra_2. We also suggest possible further improvements, that are not included in Integra_2. In Section 5.3 we give results comparing the performance of Integra_1 and Integra_2.

When solving diet problems we sometimes resolve the same problem from different random start points, to try and find the global minimum. The multiple *solves* of one problem are collectively known as a *run* of the problem. At the end of a run the solve that finds the best local minimum is presented as the solution. In Section 5.4 we give a new method for determining how many solves to do in one run of a problem.

5.1 Introduction

The GPP is similar to the standard pooling problem, therefore many of the methods applicable to solve the standard pooling problem are still relevant.

Early attempts to solve the standard pooling problem, such as in Haverly [1978], involved recursion. Complicating variables involved with the mixing bins were fixed and the resulting linear problem solved, then the complicating variables adjusted and the process repeated until convergence. This process leads naturally to sequential linear programming (SLP), used by Shell Oil since Griffith and Stewart [1961] and shortly afterwards by Exxon. Lasdon and Waren [1980]

showed SLP is superior to recursion for standard pooling problems. SLP can also be used to solve GPPs.

The standard pooling problem can be represented as a *bilinear program*. A bilinear program is a problem where the variables can be partitioned into two subsets, such that when each set is fixed the remaining problem is comparatively easy to solve. Special methods exist for solving bilinear programs. Thieu [1986] showed how a bilinear program can be converted to a *concave program*, and then solved using methods for concave programs. Audet et al. [2004] showed that the GPP can also be represented as a bilinear program.

Much of the recent effort in solving the standard pooling problem has been to find the global minimum. There are several methods for doing this, such as generalised Benders (e.g. Geoffrion [1972]), the global optimisation algorithm (Visweswaran and Floudas [1993]), spatial branch and bound (e.g. Ben-Tal et al. [1994], Quesada and Grossman [1995]) and heuristic methods (e.g. Audet et al. [2004]). Many of these methods involve solving a *linear relaxation* of the standard pooling problem, where the original problem is simplified to make it linear. In Section 5.1.3 we consider linear relaxations of the GPP, so that these methods can be used for the GPP too.

5.1.1 NLP solvers

The GPP is an NLP, hence it can also be solved by a general purpose NLP solver. An NLP can be written in the form below, where $\boldsymbol{x}$ is a general variable, $f(\boldsymbol{x})$ the (linear) objective, $h(\boldsymbol{x})$ the (nonlinear) constraints and $\boldsymbol{x} \in \boldsymbol{X}$ the simple bounds on $\boldsymbol{x}$.

$$\begin{aligned} &\min_{\boldsymbol{x}} \ f(\boldsymbol{x}) \\ &\text{s.t. } h(\boldsymbol{x}) = \boldsymbol{0}, \\ &\boldsymbol{x} \in \boldsymbol{X}. \end{aligned} \tag{5.1}$$

The table below gives the average time in seconds for solving a subset of the GPP1 problems with several different NLP solvers. The solvers are all local solvers which are not guaranteed to find the global minimum. Ipopt is an interior point solver. Lancelot and PENNON solve NLPs using augmented Lagrangians. MINOS implements a sequential linearly constrained algorithm that is similar to Lagrangian relaxation.

	Factory3	H_prob	Sonoco97	Plant1a	Sloten1relax
Ipopt	0	0	19	35	>1000
Lancelot	0	>1000	>1000	>1000	fail
Minos	0	0	6	fail	fail
Pennon	0	fail	fail	fail	fail
SQPFilter	0	1	62	91	>1000
Slp5	0	0	5	5	377

SQPFilter is a sequential quadratic programming (SQP) solver, described in Fletcher and Leyffer [1999]. Slp5 is based on SQPFilter, but instead of using quadratic approximations to the NLP it uses linear approximations. SQPFilter was first adapted to be an SLP code by Andreas Grothey, then further changes were made by the author to form Slp5. Slp5 is similar to the Integra_1 code that we describe later. The major difference is that Slp5 is a general purpose SLP solver, designed to solve any problems, whereas Integra_1 is specifically designed for pooling problems. This means Integra_1 recognises the constraint types in pooling problems and can find the gradients automatically, making it faster than Slp5 on these problems, as Slp5 always finds gradients by differentiating. An outline of an SLP algorithm is given when describing Integra_1 in Section 5.1.4.

In each case the GPP was modelled in AMPL and solved via the NEOS server. The times given are the average time of three solves. For problems that failed, or took more than 1000 seconds, this happened on all three solves. The fails were due to the solver (erroneously) declaring the problem infeasible, or just crashing. All solvers easily solved Factory3, but even the relatively small H_prob and Sonoco97 problems caused difficulties for some solvers. The large Sloten1relax problem was only solved by Slp5. Note that these solves are all done using the default settings, and it is possible that with adjustment of solve parameters the times could improve. For example, with different preprocessing to the problem data MINOS is able to solve all the problems.

The table included results for an SQP and an SLP solver. The advantage of using SQP to solve an NLP is that it enables quadratic convergence, which can be much faster than the linear convergence of SLP when the solution is not at a vertex. Fletcher and Leyffer [1997] mention that SQP performs particularly well compared to SLP in solving a system of n nonlinear equations with n unknowns, where it is essential to use second order information to get good convergence. The advantage of using SLP is faster convergence on tightly constrained problems, where the feasible region is small (Vanderplaats [1984]). In a version of SQPFilter described in Chin and Fletcher [2003], it is mentioned that for very large problems SLP can be a lot faster than SQP, partly due to the difficulty in SQP of finding global solutions to nonconvex QP subproblems, because of indefinite Hessians.

The GPPs that we solve are typically quite tightly constrained and not very nonlinear, hence SLP should perform well. This is indeed the finding of the table, and also the finding of Grothey [2001] on standard pooling problems. Grothey et al. [1999] attribute the relatively poor performance of SQP on GPPs to second order information misleading the solver.

There is some evidence that interior point may be effective for solving the LPs formed by the SLP algorithm. When solving one of the LPs for Sloten1relax (in particular, the LP whose sparsity pattern was shown in Figure (4.6) it was found that an interior point solver was significantly faster than a simplex method solver. However, the advantage of using interior point to solve each LP could be offset by the inability of interior point to effectively warm

start. A hybrid solver which solved the first few difficult SLP iterations with interior point then switched to the simplex method may outperform both methods.

We mentioned above that in one run of a problem there may be many separate solves from different random start points, in an attempt to find the global minimum. Each of these solves is almost completely independent, as the only coordination involved is in collecting the solutions and deciding when to stop doing new solves. It is therefore possible to perform several solves of one problem in parallel. Ideally when n computers are used the problem can be solved n times faster, but in practice the speedup is often less. Bolton et al. [2000] solve diet problems using a parallel computer, and report that there was very little loss in speedup. The Sloten1relax GPP was solved with a *dual core* machine, that allows two simultaneous solves. This also achieved a good speedup, of 1.9 (2.0 is perfect speed up).

5.1.2 Tightening Bounds

Before a problem is solved it is common to apply some preprocessing to the problem data. Preprocessing for Integra_1 is described in Section 5.2.1, here we give a general introduction to one important part of processing, tightening bounds. This means adjusting the simple lower and upper bounds on each variable before the solve starts. Tightening bounds can help to provide more feasible initial solution points, and can give better linearisations when solving with SLP. The reason for describing the tightening methods here, before the description of Integra_1, is that some of the tightening methods are used in the linear relaxations of the next section.

In *feasibility based bound tightening* (FBBT) bounds are tightened by considering all the equality constraints. Each equality constraint is examined and the minimum and maximum possible values of the row are compared with the row bounds, after which the row bounds can be tightened. For example if both β_{12} and y_{21} are bounded in the constraint

$$\beta_{12} y_{21} = y_{11}, \tag{5.2}$$

then the minimum and maximum possible values of the row are $\underline{\beta}_{12}\underline{y}_{21}$ and $\overline{\beta}_{12}\overline{y}_{21}$. These values can be used to tighten bounds on y_{11}. See Brearly et al. [1975] or Belotti et al. [2008] for more on FBBT. Two particular new types of FBBT are introduced below.

Network tightening is a type of FBBT that we have developed specifically for pooling problems. It uses knowledge of the network of a problem to tighten the bounds on some variables. If any of the bounds suggested by network tightening is tighter than the original bound on a variable it replaces the original bound. Network tightening can lead to substantially more feasible initial solution points. Figure 5.1 shows an example network, where as before the flow proportion bounds on input are given on each arc. Note the possibility of a negative flow proportion (drying) from Raw 1 to Mix 2. We apply network tightening to Mix 2, to determine

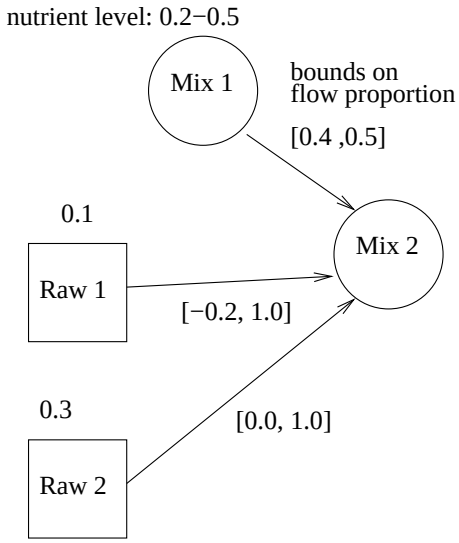


Figure 5.1: Example network

the maximum nutrient concentration that can be supplied to Mix 2. The maximum is 0.44, from taking a flow proportion of 0.5 at nutrient concentration 0.5 from Mix 1, a flow proportion of -0.2 at nutrient concentration 0.1 from Raw 1, and a flow proportion of 0.7 at nutrient concentration 0.3 from Raw 2.

Large bound tightening is also a type of FBBT specifically introduced here for pooling problems. Very large (possibly infinite) bounds are replaced with reasonable bounds. This tightening is necessary on our test problems, as in some of them the raw masses have bounds of ±999999999 (effectively infinite), and the mix masses and mix costs are usually not given bounds at all.

	lower bound	upper bound
nutrient n	0	$2 \max_r X_{rn}$
mass	0	$2 \sum_{p=1}^{P} V_p$
cost	0	$2 \max_r C_r$

We suggest new bounds for the pooling problem that are often quite loose, but are often still substantially tighter than the original bounds. As an example the table above shows the new lower and upper bounds suggested for mix m. If any of these bounds is tighter than the original bound on a variable they replace the original bound. For the nutrient concentration of nutrient n in the mix the upper bound is twice the maximum nutrient concentration in any raw. For the mass of the mix the upper bound is twice the sum of all the product masses. For the cost of the mix the upper bound is twice the cost of the most expensive raw. All the lower bounds are set to zero. The reason that large bound tightening gives upper bounds above what might be considered the maximum possible, is to allow for drying. Although the maximum level of drying observed in the test problems is 20%, here we conservatively cater for up to 100% drying.

Optimality based bound tightening(OBBT) gets bounds on each variable in turn by solving a linear relaxation of the problem with the objective each time to minimise or maximize that variable. It is described in e.g. Nowak and Vigerske [2007] or Belotti et al. [2008]). In one iteration of OBBT two LPs are solved for each variable, one maximising and one minimising, and for each LP solve the tightest bounds found so far are used on all variables. In multiple iterations of OBBT the bounds can be progressively reduced.

Just one iteration of OBBT is hugely time consuming, as it requires solving two LPs for every variable. In Nowak and Vigerske [2007] OBBT is only performed for certain variables where it is expected to be most beneficial. The method applied here is not intended to be a practical presolve, just an effort to find bounds that are as tight as possible.

	Factory3	H_prob	Sonoco97	G1b	Plant1a
variables	34	592	782	608	3582
bounds moved by OBBT	48	641	392	892	5220
average bound distance	0.346	0.120	0.504	0.064	0.159

We now show that OBBT is effective on GPPs. The table above shows the effect of one iteration of OBBT, applied to a linear relaxation of the pooling problem (specifically linear relaxation $rl5$, described later, which already has some FBBT). The first row of the table shows the number of variables in each problem. The second row shows the number of bounds (lower and upper bounds) that were improved by one iteration of OBBT. The number of bounds moved varied between an average of 1.47 bound moves per variable in G1b, to only 0.50 bound moves per variable in Sonoco97. We refer to the *average bound distance* as the average distance between lower and upper bounds among all (scaled) variables in a problem. The third row of the table shows the ratio of average bound distance after one iteration of OBBT to before. The ratios are significantly less than one, indicating the average bound distances have fallen greatly. Thus OBBT is effective, though as stated above is not practical except in very rare cases.

5.1.3 Linear relaxations

Linear relaxations are formed and solved in several of the algorithms that solve the standard pooling problem. If the linear relaxation is sufficiently similar to the original NLP, the objective value at the LP solution can give a good lower bound on the NLP objective value, and the linear relaxation solution can be used as a warm start point for the NLP.

Two types of linear relaxation have been applied to the standard pooling problem. In *Lagrangian relaxation* complicating constraints, in this case the nonlinear constraints, are relaxed, and added to the objective function. This results in a model that is still nonlinear. In *linear programming relaxation* each nonlinear constraint is replaced by its linear convex hull to give a linear model (see e.g. Foulds et al. [1992] or Meyer and Floudas [2006]. This resulting model can be strengthened by multiplying together linear constraints and bounds to form new (redundant)

constraints, then adding the convex hull of some of these new constraints to the model (see e.g. Sherali and Alameddine [1992], Liberti and Pantelides [2006]). For the pooling problem the lower bound from Lagrangian relaxation is often tighter than the lower bound from linear programming relaxation (see e.g. Ben-Tal et al. [1994], Adhya and Tawarmalani [1999], Almutairi [2008]). However Tawarmalani and Sahinidis [2002] showed that the linear programming relaxation of the pooling problem modelled with the *pq*-formulation (see Section 4.1.3) gives even tighter bounds than the Lagrangian relaxation.

As GPPs can have multiple layers of mixing bins they can have more nonlinearity than the standard pooling problem, and it is expected that finding tight linear relaxations will be harder. Below we introduce several different linear relaxations for the GPP. The best of these, $rl5$, is fairly tight on all test problems.

We first consider a linear relaxation where the mixes, and their corresponding nonlinear constraints, are removed from the model. However unlike Lagrangian relaxation the nonlinear constraints are not placed in the objective. Instead a series of new linear constraints is added to the model, limiting the flow from raws to products. These constraints attempt to capture the limitations imposed by the nonlinear constraints, but are of course not as tight and are only outer approximations. We call this *Lagrangian-style* relaxation, as it has in common with Lagrangian relaxation that the complicating constraints are removed from the model. In our first linear relaxation, called $rl3$, the mixes are omitted and flow from raws to products is only permitted where flow was possible in the original model, and only at levels that were possible.

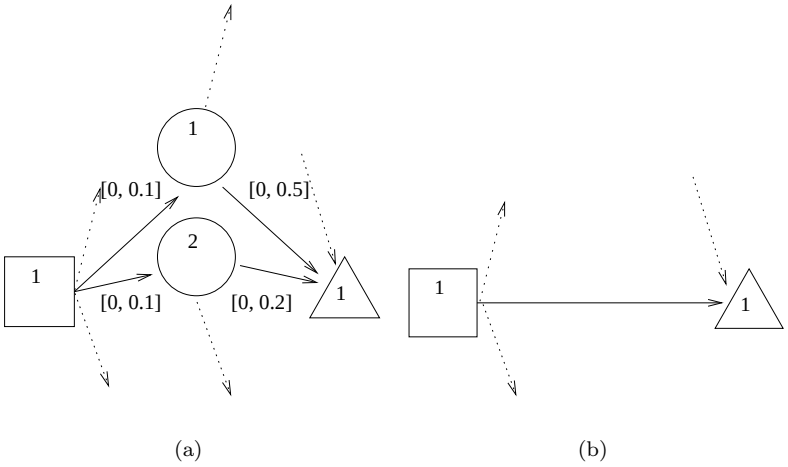


Figure 5.2: Nonlinear network (a) and linear network after relaxation (b)

Figure 5.2 shows how a nonlinear network becomes a linear network with the removal of the mixes. In $rl3$ flow would be permitted between Raw 1 and Product 1 with bounds of $[0, 0.07]$. This 0.07 upper bound is calculated using the bounds on flow proportion in the original model, where there is flow from Raw 1 to Product 1 via Mix 1 and via Mix 2, at a maximum total rate of $0.1(0.5) + 0.1(0.2) = 0.07$.

In *rl4a* the mixes are retained, and a linear relaxation is formed by replacing each nonlinear constraint by its convex hull. This is therefore a linear programming relaxation. In order to find these convex hulls the problem is first reformulated, in a method similar to Belotti et al. [2008]. To reformulate the problem extra variables are introduced that are each equal to one of the bilinear terms in a nonlinear constraint. As an example consider the product nutrient constraint (4.8) for Product 1 and Nutrient 1:

$$\sum_{r=1}^{R} \gamma_{1r} X_{r1} + \sum_{m=1}^{M} \delta_{1m} y_{m1} = z_{11}. \tag{5.3}$$

This says that the Product 1 concentration of Nutrient 1 is the total of all the inputs from raw materials to the product and all the inputs from mixes to the products. In the reformulation we define new variables for the bilinear terms:

$$\sum_{r=1}^{R} \gamma_{1r} X_{r1} + \sum_{m=1}^{M} z^{lin}_{1m1} = z_{11}, \tag{5.4}$$

$$z^{lin}_{1m1} = \delta_{1m} y_{m1}, \qquad m \in 1 \ldots M. \tag{5.5}$$

Constraint (5.4) is now linear, as all the nonlinearity is contained in the simple bilinear constraint (5.5). Reformulations like this are applied to all nonlinear constraints, so that all the nonlinearity is contained in simple bilinear constraints. These have a straightforward convex hull, given by four linear constraints known as the McCormick relaxation, see McCormick [1983]. For (5.5) the linear constraints are (omitting subscripts and superscripts):

$$\begin{aligned} z &\geq \delta \underline{y} + \underline{\delta} y - \underline{\delta}\,\underline{y}, \\ z &\geq \delta \overline{y} + \overline{\delta} y - \overline{\delta}\,\overline{y}, \\ z &\leq \delta \underline{y} + \overline{\delta} y - \overline{\delta}\,\underline{y}, \\ z &\leq \delta \overline{y} + \underline{\delta} y - \underline{\delta}\,\overline{y}. \end{aligned} \tag{5.6}$$

where $\underline{\delta}$, $\overline{\delta}$, $\underline{y}$ and $\overline{y}$ are lower and upper bounds on δ and y. The first two constraints define two linear faces beneath the nonlinear constraint, and the second two constraints define two linear faces above the constraint. The convex hull relaxation is then strengthened with the addition of the linear constraint below:

$$\sum_{r=1}^{R} t_r = \sum_{p=1}^{P} V_p. \tag{5.7}$$

This constraint says that the total mass of raws must be equal to the total mass of products. It makes the relaxation considerably tighter, without having any significant impact on the solve time.

In $rl5$ the linear constraints from $rl3$ are used in addition to the linear constraints of $rl4a$, combining the Lagrangian-style relaxation with the linear programming relaxation. This creates a relaxation tighter than either of them.

In $rl3$, $rl4a$ and $rl5$ the bounds on each variable are also tightened, which makes the relaxations tighter. Bounds on variables are tightened using both network tightening and large bound tightening. These are both described above in Section 5.1.2. In $rl6$ the linear relaxation of $rl5$ is further strengthened by applying one iteration OBBT. OBBT is also described in Section 5.1.2.

	Factory3	H_prob	Sonoco97	Global_data	G1b	Plant1a	Sloten1relax
$rl3$	0.968	0.699	0.998	0.856	0.954	0.928	0.866
$rl4a$	0.963	0.925	0.229	0.810	0.857	0.884	0.395
$rl5$	0.990	0.994	0.998	0.951	0.996	0.985	0.949
$rl6$	0.998	1.000	0.998		1.000	1.000	

The linear relaxations have more constraints then the original NLP, but can be solved by one LP solve rather than with the many LP solves of SLP. They are therefore typically faster than an original NLP solve.

The table above gives the ratio of the objective value of the linear relaxation to the objective value of the original NLP. The gaps in the able occur as relaxations $rl6$, which involves OBBT, was only applied to some of the small test problems. Neither of $rl3$ and $rl4a$ dominates the other. Relaxation $rl5$, which combines constraints from the Lagrangian-style relaxation $rl3$, and the linear programming relaxation $rl4a$, is of course tighter than both, and had ratios close to 0.95, or greater, for all problems. Although where tested they are even tighter $rl6$ takes an impractically long time, so $rl5$ is the recommended linear relaxation.

Using as warm start

The best of the linear relaxations was $rl5$, as it was the tightest relaxation that did not take an excessive time. We now look in more detail at the time to solve this linear relaxation, compared to the total time to solve an SLP, and whether it is practical to solve GPPs by warm starting from the solution to $rl5$.

The first row of the table below shows the ratio of initial infeasibility of an SLP warm started with the solution to linear relaxation $rl5$, to the average initial infeasibility of SLPs warm started

	Factory3	H_prob	Sonoco97	Global_data	G1b	Plant1a	Sloten1relax
initial inf	0.063	0.008	0.009	0.001	0.005	0.015	0.030
LP iterations	0.802	0.667	0.807	0.703	0.985	0.766	0.380

from random points. The second row shows the ratio of LP iterations of the warm started SLP to the random started SLPs. The table shows that when using the warm starts the initial infeasibility is significantly less for all problems. With the warm start all problems also solve in fewer LP iterations. However when the time to solve the linear relaxation is taken into account only the large Sloten1relax problem benefits from this warm start.

5.1.4 Integra_1

We now introduce a specialised GPP solver called Integra_1. Integra_1 is an SLP solver based on the SQPFilter solver of Fletcher and Leyffer [1999], that uses linear approximations to the NLP instead of quadratic approximations. Integra_1 is currently used by Format International to solve GPPs. It is similar to the SLP solver Slp5, but is specialised to recognise the constraints of pooling problems so it can find the derivatives immediately. Integra_1 solves GPPs that are modelled with a formulation very similar to the extended q-formulation, that was described in Section 4.2.3. The one difference is that in Integra_1 the objective function is defined not as the sum of the costs of the raws, but as the sum of the costs of the products. This requires a few extra variables and nonlinear constraints to calculate the cost of the products. However, in Integra_1 the variables and nonlinear constraints to define the masses of the raws are only included when needed (when a raw mass is bounded), so depending on the particular problem either formulation can be larger. Before the solve begins there is some preprocessing of the problem data. Changes to the preprocessing of Integra_1 are discussed in Section 5.2.1.

1	NLP linearised about trial point, and trust region added.
2	Resulting LP minimised to give a trial solution point, with objective value (in the original NLP) of f and infeasibility of h.
3	If (f,h) values are *acceptable* a *step* is taken to a new trial point.
4	Trust region size varied depending on last LP solve.

Table 5.1: An iteration of the Integra_1 SLP algorithm

Table 5.1 gives an outline of an iteration of the SLP algorithm employed in Integra_1. In Step 1 the NLP is linearised about an initial point, using the Taylor expansion, and in Step 2 this LP is minimised. We assume for now that the LP is feasible. We consider what happens when an LP is infeasible later on in this section. The linearisation is only expected to closely model the NLP close to the point of linearisation. Hence it is common in an SLP algorithm to limit the size of the step, either with a *trust region* or *linesearch*. A trust region is a set of constraints that limit the maximum movement by any variable each LP solve. In a linesearch the LP is solved and then various points are sampled along a line from the old LP solution point to the new LP solution point, and a step taken to the best one. In Integra_1 a trust region is

used. Changes to the trust region of Integra_1, and the possible introduction of a linesearch, are discussed in Section 5.2.2.

Each solution point $\boldsymbol{x}$ has an NLP objective, $f(\boldsymbol{x})$ or simply f, and an NLP infeasibility, $h(\boldsymbol{x})$ or simply h. Any point with small enough infeasibility (in practice below 10^{-5}), is called *feasible*.The solution point is tested for how close f and h are to the f and h values predicted by the LP solve. This is called the *ratio test*. If the linearisation is locally accurate then f and h will be close to those predicted by the LP, and the ratio test will be passed. Changes to the ratio test of Integra_1 are discussed in Section 5.2.3.

Ideally successive iterations will lower both objective value f and infeasibility h, though this does not always happen. Integra_1 employs a *filter*, where each LP solution point is either *accepted* or *rejected* depending on the values of f and h. A point is said to be *dominated* by another point if that other point has both lower f and lower h. If a point is not dominated by any other point it is accepted by the filter, else it is rejected. In Step 3 if the new point is accepted a step is made from the old point to the new point and a linearisation formed about the new point. If the new point is accepted, and the ratio test is passed, and at least one variable has moved to the edge of the trust region boundary (implying that the previous trust region was active), then in Step 4 the trust region is enlarged, by doubling it. If the new point is rejected then for the next LP solve the same linearisation is used and the trust region is shrunk, to one quarter of the largest movement in any variable in the last LP solve. See e.g. Fletcher and Leyffer [1997] for more on filters. Changes to the filter of Integra_1 are discussed in Section 5.2.4.

We now consider what happens if the LP is infeasible. The iterations described above are discontinued and a new problem is solved to find a feasible point. This is also done by an SLP filter algorithm. The Integra_1 algorithm therefore consists of two types of iteration. We use the term *phase one* to refer to an SLP iteration that has the primary objective of reducing infeasibility, and *phase two* to refer to an SLP iteration that has the primary objective of reducing the objective value. Integra_1 begins with a phase two iteration, and remains in phase two as long as each LP is feasible. The details of the phase one SLP algorithm are similar to those of phase one.

$$\begin{aligned} \min_{\boldsymbol{x}} \ & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t. } & \boldsymbol{l} \leq \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{u}, \\ & -\rho\boldsymbol{s} \leq \boldsymbol{x} - \hat{\boldsymbol{x}} \leq \rho\boldsymbol{s}, \\ & \boldsymbol{x} \in \boldsymbol{X}. \end{aligned} \tag{5.8}$$

Recall the NLP model of (5.1). In phase two we solve a linearisation of this given by the model

above. The objective is the general variable $\boldsymbol{x}$, multiplied by the costs $\boldsymbol{c}$. The first constraint says that the constraint matrix $\boldsymbol{A}$ multiplied by $\boldsymbol{x}$ must be between the bounds of $\boldsymbol{l}$ and $\boldsymbol{u}$. The second constraint applies a trust region of size ρ. $\boldsymbol{s}$ is a vector of scaling parameters and $\hat{\boldsymbol{x}}$ the last value of $\boldsymbol{x}$, so the constraint forces $\boldsymbol{x}$ to be within $\rho\boldsymbol{s}$ of its last value. The final constraint applies simple bounds on $\boldsymbol{x}$. The solution to this LP has objective value that estimates the true NLP objective of f. At the solution to the LP we estimate that the infeasibility $h = 0$, as we assume being LP feasible implies the NLP is close to feasible.

$$
\begin{aligned}
&\min_{\boldsymbol{x},\boldsymbol{y}} \boldsymbol{e}^T\boldsymbol{y} \\
&\text{s.t. } \boldsymbol{l} \leq \boldsymbol{A}\boldsymbol{x} + \boldsymbol{I}\boldsymbol{y}, \\
&\qquad \boldsymbol{A}\boldsymbol{x} - \boldsymbol{I}\boldsymbol{y} \leq \boldsymbol{u}, \\
&\qquad -\rho\boldsymbol{s} \leq \boldsymbol{x} - \hat{\boldsymbol{x}} \leq \rho\boldsymbol{s}, \\
&\qquad \boldsymbol{x} \in \boldsymbol{X}, \\
&\qquad \boldsymbol{y} \geq \boldsymbol{0}.
\end{aligned}
\tag{5.9}
$$

In phase one the original problem is reformulated to have objective of minimising constraint violation. The linearised phase one problem is given above. $\boldsymbol{y}$ is a new vector variable, known as an artificial variable, which measures the amount of constraint violation. In the objective it is weighted by $\boldsymbol{e}^T$, a vector of ones, so that the goal of the phase one problem is to minimise the sum of the artificial variables. This is achieved when $\boldsymbol{y} = \boldsymbol{0}$.

In the constraints $\boldsymbol{I}$ is an identity matrix. The first constraint on $\boldsymbol{x}$ in the phase two model of (5.8) has been relaxed into two inequalities in (5.9). If neither of these two inequalities can be satisfied normally then part of the vector $\boldsymbol{y}$ must become nonzero, increasing the objective value. The solution to this LP shows how infeasible the LP is, and so estimates h. Once phase one is successful the SLP returns to phase two. Changes to phase one of Integra_1 are discussed in Section 5.2.5.

Integra_1 solves all the LPs using EMSOL (Hall [1997]). EMSOL is a revised simplex method solver that uses Harris Devex pricing. Each iteration of EMSOL is referred to as an *LP iteration*. Changes to the LP solve of Integra_1 are discussed in Section 5.2.6.

The phase two iterations terminate once a suitable solution point has been found, or the trust region becomes sufficiently small. Changes to the convergence criteria of Integra_1 are discussed in Section 5.2.7.

5.2 Developing Integra_2

Integra_2 is our improved version of Integra_1. It has changes to Integra_1 in preprocessing, the trust region, the ratio test, the filter, phase one, the LP solver and convergence criteria. In the following subsections we discuss each of these seven areas. In each subsection we begin by outlining what is done in Integra_1, then describe the changes implemented in Integra_2. In some cases we then suggest further improvements, which have not yet been included in Integra_2.

5.2.1 Preprocessing

Before a problem is solved it is common to apply preprocessing, to make the problem easier to solve. This is especially important for a difficult nonlinear problem like the pooling problem. Two common methods of preprocessing are scaling and bound tightening. Bound tightening has already been introduced in Section 5.1.2. Scaling a problem can reduce numerical errors, and when a trust region is used can reduce the number of iterations taken. In Integra_1 instead of directly scaling the coefficients on variables in the constraint matrix, the size of the trust region in the direction relating to each variable is scaled. The scaling is such that a movement in any variable equal to the width of its trust region causes a similar change in the constraints. The bounds on a variable x_i are such that:

$$-s_i\rho \leq x_i - \hat{x}_i \leq s_i\rho, \tag{5.10}$$

where ρ is the trust region size, $\hat{x}_i$ the last value of x_i and s_i the scaling parameter on the variable. For each nutrient concentration variable s_i is set to the average concentration of that nutrient among all the raws. For all mass variables s_i is set to the average mass of the products. For all cost variables s_i is set to the average cost of the raws. For all flow variables s_i is set to 1. There is also scaling in Integra_1 by the LP solver, as at the start of each LP solve EMSOL performs row and column scaling. This is very effective in improving the solution time to the LP problems.

Integra_1 tightens the bounds on some variables before starting the solve. There is FBBT on some, but not all, the constraints. In Integra_1 the FBBT only tightens bounds on variables which are the right hand side of equality constraints, such as y_{11} in (5.2). Examining the model constraints in (4.6)-(4.11) reveals that the flow proportion variables do not appear as right hand sides of equality constraints, hence they are not tightened in Integra_1. Furthermore the Integra_1 FBBT goes through the constraints in the order that they appear in the model, so if, for example, a bin labelled with a high number flows into a bin labelled with a low number, tightening does not occur.

As well as scaling and bound tightening a third aspect of preprocessing is the selection of the initial solution point. In Integra_1 the initial solution point is selected by uniformly randomly choosing a value for each variable between that variable's bounds. However, to account for cases where a variable is not bounded at all, initial values for all variables are forced to be within M of the origin, where M is a given large integer (and $\boldsymbol{M}$ a vector with all elements equal to M). Let $\boldsymbol{x}$ again be a general variable with bounds $\underline{\boldsymbol{x}} \leq \boldsymbol{x} \leq \overline{\boldsymbol{x}}$, and let $\boldsymbol{p}$ be a vector of uniform random variables, with $\mathbf{0} \leq \boldsymbol{p} \leq \mathbf{1}$. Then the initial solution point for $\boldsymbol{x}$ is given by:

$$\hat{\boldsymbol{x}} = \boldsymbol{p}^T \max(\underline{\boldsymbol{x}}, -\boldsymbol{M}) + (\mathbf{1} - \boldsymbol{p})^T \min(\overline{\boldsymbol{x}}, \boldsymbol{M}), \tag{5.11}$$

where the min and max operations are done for each component of the vectors. In Integra_1 M is set to 10,000, meaning that all variables are initialised with values between -10,000 and 10,000.

Integra_2

We now consider improvements to the preprocessing that go into Integra_2. In Integra_2 as well as the FBBT described above for Integra_1 we also perform large bound tightening, as described in Section 5.1.2.

The effect of the changes to the preprocessing of Integra_1, combined with all the other changes, is considered when we compare Integra_1 and Integra_2 in Section 5.3.

Further improvements

We now consider further improvements to the preprocessing of Integra_1. At present these are not included in Integra_2.

The constraints (4.6), (4.8), (4.10) and (4.11) are all equality constraints that define a variable, which can be eliminated from the model. For some problems, such as Sloten1relax (see Figure 4.6), such an elimination drastically reduces the problem size. Eliminating all these constraints means the only variables left are flow proportion variables, which are enough to completely describe the solution. These variables are automatically bounded between zero and one (or slightly wider for a GPP where drying is possible), so when all variables are flow proportion variables no scaling is needed.

Recall that in Integra_1 the initial solution point is selected using (5.11). Using this formula after first eliminating some variables, as described above, leads to more feasible initial solution points. In (5.11) the initial value of each variable is generated independently. For an equality constraint, such as (5.2), it is better to fix β_{12} and y_{21} say, and then solve for y_{11}. The resulting value for y_{11} may still be outside its bounds, but at least the equality constraint is satisfied.

Using a smaller value of M in (5.11) can give more feasible initial solution points. For all GPPs tested using $M = 1$ gave more feasible initial solution points and a typical saving in solution time of 20%. For the problem Sloten1relax using $M = 1$ had a drastic effect, saving 90% of solution time. However, there are concerns that using a small value of M means all the randomly generated start points are close to each other, and multiple solves will not find a full range of local minima. This would need to be further investigated before adopting a smaller value of M in Integra_2.

We showed above in Section 5.1.3 that we can solve a linear relaxation to give a warm start point that is close to feasible. We can also heuristically find good values for the nutrient concentration variables by considering the bounds on nutrient concentration in the mixes and products. Suppose, for example, that some raws have a very high concentration of nutrient n, but most products have low requirements of that nutrient. Although it is possible to supply a mix with a high concentration of the nutrient, that is unlikely to be optimal. A heuristic that initialises mix nutrient concentrations based on this observation was able to find good initial solution points.

5.2.2 Trust region

In Integra_1 there is a trust region of size ρ that limits the movement in any variable in each LP solve, as shown in (5.10). The initial size of the trust region is $\rho = 5$. The trust region can shrink or grow between LP solves, though we do not treat any variables specially and they all get given the same size (scaled) trust region.

Integra_2

In Integra_2 we reduce the effect on the trust region for the first few iterations, allowing the LP to take larger steps. This is done by having an initial trust region size of $\rho = \rho^{max}$, defined as:

$$\rho^{max} = \max_i \frac{\overline{x}_i - \underline{x}_i}{s_i} \quad i \in \text{ set of variables}, \tag{5.12}$$

so ρ^{max} is large enough that so that all variables can move all the way between their bounds (hence with $\rho = \rho^{max}$ the trust region is inactive). $\boldsymbol{x}$, is a general variables with bounds $\underline{\boldsymbol{x}}$ and $\overline{\boldsymbol{x}}$ and $\boldsymbol{s}$ is the scaling parameter. When the trust region is shrunk it is still reduced to one quarter of the largest movement in any variable in the last LP solve, so however large ρ^{max} is the trust region can still quickly become small, and starting with $\rho = \rho^{max}$ does not inhibit convergence of the algorithm.

	f	h	$\lvert d\rvert$	LP iterations
Initial $\rho = 5$	2205	1423	5	4564
Initial $\rho = \rho^{max}$	1204	322	143	4888

The table above compares the SLP objective value, f, infeasibility, h, largest movement in any variable in last LP solve, $|d|$ and LP iterations so far, after two SLP iterations of a solve of Sloten1relax. The first row is the solve beginning with $\rho = 5$, as in Integra_1. The second row is the solve beginning with $\rho = \rho^{max}$. With $\rho = \rho^{max}$ the objective value and infeasibility are both considerably better than the solve beginning with $\rho = 5$, with a comparable amount of LP iterations. This is typical, starting with a larger trust region means larger steps are taken and feasible points are found more quickly. There is a danger that given a large initial trust region the same local minimum will be found regardless of the initial solution point, so less of the solution area will be explored, and so the global minimum will be less likely to be found. This was not found to be the case though (as we discuss in the results of Section 5.3).

Further improvements

We now consider possible further improvements to the use of the trust region by Integra_1. These are not included in Integra_2. We mentioned above that SLP algorithms commonly use *either* a trust region *or* a linesearch method. We now consider adding a linesearch to Integra_1 on top of the existing trust region.

A linesearch evaluates the f and h values for several points along the direction from the old solution point to the proposed new solution point. These evaluations are simple to do, and may find a better solution point. Let $\hat{\boldsymbol{x}}$ be the old solution point, $\boldsymbol{d}$ be the proposed step and p the proportion of the step to take. New points, $\boldsymbol{x}$, are generated by varying p in the equation below.

$$\boldsymbol{x} = \hat{\boldsymbol{x}}x + \boldsymbol{d}p, \qquad p \in P, \tag{5.13}$$

where P is the set of possible step sizes to choose. Choosing $p = 1$ corresponds to taking the standard solution point, other values of p correspond to taking a different solution point. Recall one point is said to *dominate* another point if it simultaneously has better f and h values. Figure 5.3 shows the points generated by a linesearch during an iteration of Sonoco97, with $P = \{0.4, 0.8, 1.0, 1.2, 1.6, 2.0\}$. The old solution point (which corresponds to $p = 0$) is also shown. Although none of the points in P dominate the old point, which has the best objective value, one point (p=0.8) dominates the standard solution point (p=1.0).

When using a linesearch a method is needed to select which point to adopt, based on the f and h values of the points. In the following test we use a conservative rule that adopts a point with p other than 1 only if that point dominates all others found by the linesearch.

The table shows the effect of the linesearch on a solve of Sonoco97 by Integra_1. For each solve we give the number of SLP and LP iterations in phase one, the final objective value and infeasibility at the end of phase one, and the number of SLP and LP iterations in phase two.

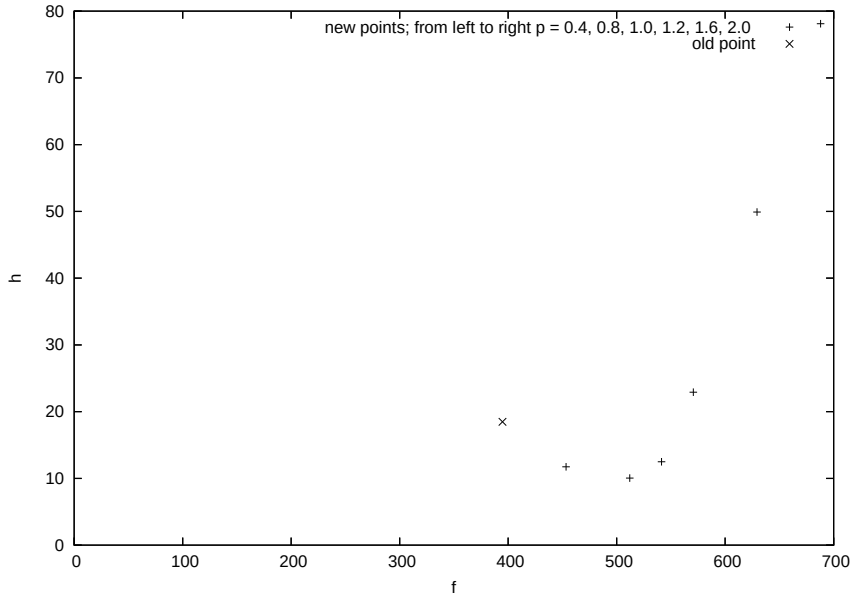


Figure 5.3: Points generated by linesearch on an iteration of Sonoco 97

	Phase one				Phase two	
	SLP its	LP its	f	h	SLP its	LP its
Standard solve	7.5	1192.0	4761	78.8	92.0	7682
Linesearch	7.7	1216.7	4557	50.9	40.7	5516

The numbers in the table are the averages of ten solves from different random start points. In phase one the two solves take about the same number of SLP and LP iterations, though the solve with a linesearch finishes phase one with a more feasible point, and then takes fewer iterations in phase two. This suggests the addition of a linesearch could be beneficial.

5.2.3 Ratio test

One of the conditions for increasing the trust region size is that the ratio test is passed. The ratio test determines how well the LP model matched the NLP in the last LP solve. In Integra_1 phase one the ratio test is passed if the infeasibility reduction achieved in the NLP is sufficiently similar to the infeasibility reduction predicted by the LP. In phase two the ratio test is passed if the reduction in infeasibility, and the reduction in objective value, achieved by the NLP are sufficiently similar to the reductions predicted by the LP.

Integra_2

In Integra_2 the ratio test in phase two is altered so that, if a point is feasible (recall any point with infeasibility below 10^{-5} is considered feasible), only the reduction in objective value is considered. This prevents a point failing the ratio test when the reduction in the objective value is good, but the infeasibility is increased from one feasible point to another feasible point. We also correct a sign error and numerical error in the ratio test of Integra_1.

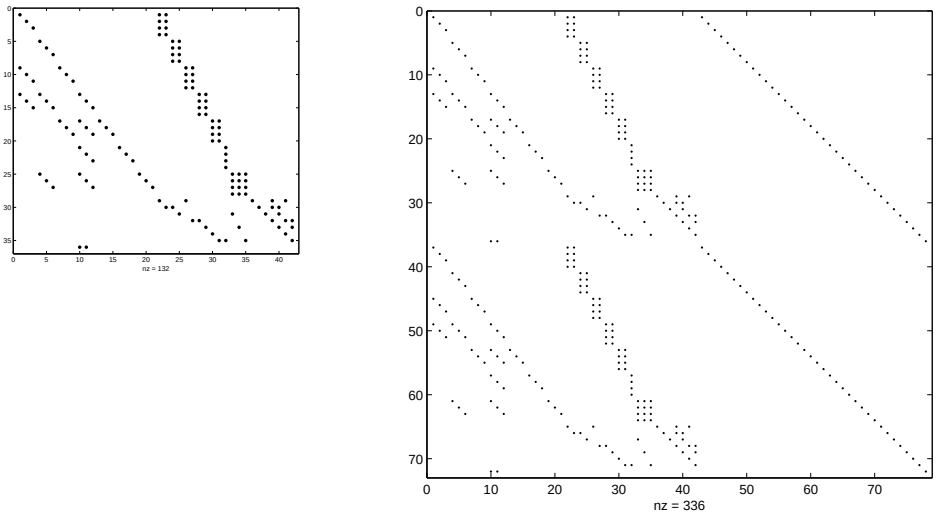
5.2.4 Filter

Integra_1 employs a filter to control the size of the trust region. Each LP solution point has its f and h values compared to the other points in the filter and if it is not dominated by any of them it is accepted, else it is rejected. If a point is accepted, and other conditions are met, the trust region is enlarged. If the point is rejected the trust region is shrunk.

Integra_2

In Integra_2 we strengthen the ratio test to reject a feasible point if there exists any other feasible point with better objective, regardless of whether or not that other feasible point has lower infeasibility. Rather like the change to the ratio test above, we are removing the distinction between the infeasibility of feasible points. This means fewer points are accepted. This can save iterations towards the end of a solve, where instead of accepting a series of points that slightly decrease the infeasibility of an already feasible point, a point is rejected and the trust region is shrunk.

5.2.5 Phase one



(a) (b)

Figure 5.4: Factory3 Integra_1 phase two LP constraint matrix (a) and Integra_1 phase one LP constraint matrix (b)

Consider the phase one problem solved by Integra_1, given in (5.9). This model has two copies of the constraint matrix $\boldsymbol{A}$. Where $\boldsymbol{A}$ has m rows and n columns the model (5.9) has $2m$ rows and $n + m$ columns. This makes it considerably larger than the phase two problem. Figure 5.4 shows the increase in size from the phase two LP constraint matrix to the phase one LP constraint matrix, for the problem Factory3.

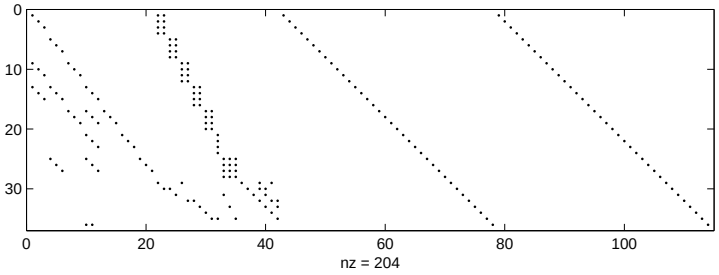


Figure 5.5: Integra_2 phase one LP constraint matrix for Factory3

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}} \ & \boldsymbol{y} + \boldsymbol{z} \\ \text{s.t. } & \boldsymbol{l} \leq \boldsymbol{A}\boldsymbol{x} - \boldsymbol{I}\boldsymbol{y} + \boldsymbol{I}\boldsymbol{z} \leq \boldsymbol{u}, \\ & \boldsymbol{x} \in \boldsymbol{X}, \\ & \boldsymbol{y}, \boldsymbol{z} \geq \boldsymbol{0}. \end{aligned} \tag{5.14}$$

Integra_2 uses a more compact model for the phase one problem, given above. $\boldsymbol{y}$ and $\boldsymbol{z}$ are artificial variables and as before $\boldsymbol{I}$ is an identity matrix. This model is equivalent to the Integra_1 phase one, but is easier to solve. Figure 5.5 shows the Integra_2 phase one constraint matrix for Factory3, which has just m rows and $n + 2m$ columns. Another improvement in Integra_2 is to warm start the first iteration of phase one, calculating the basic or nonbasic status of the columns of (5.14) based on the last phase two solution.

5.2.6 LP solver

In Integra_1 most of the time spent solving a GPP is in solving the LPs, using EMSOL. Like the SLP algorithm the LP has two types of iteration, which we call *LP phase one* and *LP phase two*. LP phase one attempts only to make the problem feasible, with no consideration of objective value. Once a feasible point is found EMSOL switches to LP phase two, which minimises the objective value while maintaining feasibility.

Integra_2

In Integra_2 we alter LP phase one so that as well as seeking a feasible point a good objective value is sought. This is done by adding a (scaled) multiplier μ of the phase two costs into the phase one objective. Thus the objective of (5.14) is replaced by $\min_{\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}} \boldsymbol{y} + \boldsymbol{z} + \mu \boldsymbol{c}^T \boldsymbol{x}$, where $\boldsymbol{c}$ are the original objective costs of the phase two model. We use a small value of μ, meaning that the priority in LP phase one is still to find a feasible point. However adding in a multiplier

of the original objective can mean that phase one finishes with a feasible point with better objective value, and so reduce the overall number of LP iterations.

5.2.7 Convergence

Suppose we have a general NLP as in (5.1). The *Lagrangian* of the objective function, $L(\boldsymbol{\lambda})$, and its derivative with respect to $\boldsymbol{x}$, are given by:

$$L(\boldsymbol{\lambda}) = f(\boldsymbol{x}) - \boldsymbol{\lambda_r} h(\boldsymbol{x}) - \boldsymbol{\lambda_c} \boldsymbol{x}, \tag{5.15}$$

$$\boldsymbol{\nabla_x} L(\boldsymbol{\lambda}) = \boldsymbol{\nabla_x} f(\boldsymbol{x}) - \boldsymbol{\lambda_r} \boldsymbol{\nabla_x} h(\boldsymbol{x}) - \boldsymbol{\lambda_c}, \tag{5.16}$$

where $\boldsymbol{\lambda_r}$ are the row duals, from $h(\boldsymbol{x}) = 0$, and $\boldsymbol{\lambda_c}$ the column duals, from $\boldsymbol{x} \in \boldsymbol{X}$. These duals can be estimated from the last LP solve. $\boldsymbol{\nabla_x} L(\boldsymbol{\lambda})$ is referred to as the *KT residue*. Under the assumption that *constraint qualification* holds any feasible point that has zero KT residue is a stationary point (Winston [2000]), which in practice when minimising means a local minimum.

The convergence criteria for Integra_1 are that both the current point is feasible, and that either the KT residue has fallen below 10^{-6} (at which point a local minimum is declared), or that the trust region is sufficiently small (at which point we stop anyway).

Integra_2

In Integra_2 we alter the definition of the KT residue in (5.16), so that we do not include column duals $\boldsymbol{\lambda_c}$ from variables that are at the trust region boundary. This means that if the last LP solve had variables pushing against the edge of the trust region the KT residue is nonzero, so that point cannot be declared a local minimum.

In Integra_2 rather than terminate with a feasible point with KT residue below 10^{-6}, we terminate when the KT residue falls below the parameter K^{res}. K^{res} has a value proportional to the size of the largest objective cost, and is often larger than 10^{-6}. Using a larger tolerance on the KT residue can save SLP iterations, and was not found to adversely effect the accuracy of the SLP solutions.

5.3 Integra_2 results

The Integra_2 code is based on Integra_1. It has all the improvements described in the Integra_2 subsections above, but not those described in the further improvement subsections. Some of the improvements to Integra_1 have been added because they improved the solution time when added individually, some were added as they give theoretical improvements. We now show that our solver Integra_2 is an improved version of Integra_1.

	Sonoco97	Sept98	Jan99	Gns	Nville	Sloten1relax	Hen040304
Successful solves	20	18	19	13	19	20	20
BestObjVal	296.451	122309	186356	119017	2925.3	1848576	106242
Average SLP its	65.5	129.0	115.8	263.3	151.3	18.7	18.9
Average time	3.7	10.9	14.6	17.5	74.1	371.7	563.0
Successful solves	20	18	20	17	19	17	18
BestObjVal	296.451	122304	186097	119023	2925.4	1848576	106242
Average SLP its	53.1	122.0	87.5	238.0	106.2	16.8	7.9
Average time	2.3	7.3	10.5	13.3	67.8	229.9	416.2

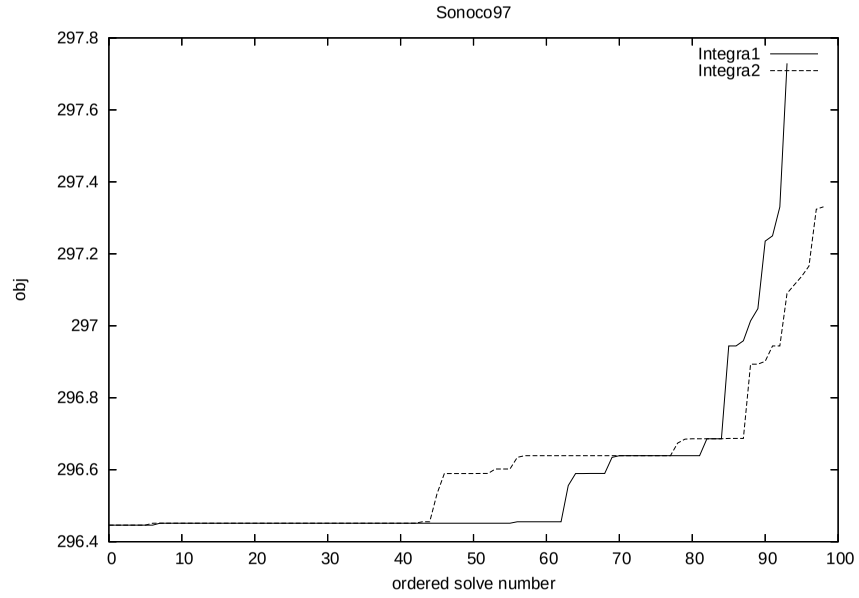
Table 5.2: Results for Integra_1 (first four lines) then Integra_2 (second four lines)

Table 5.2 compares Integra_1 and Integra_2 on the test set GPP2. This shows the combined effect of using all the improvements. Each problem is solved 20 times from different random start points with each solver. The first four rows of the table are for Integra_1. The first row of the table gives the number of successful solves, out of 20. The second row gives the best objective value found in any of the solves. The third row gives the average number of SLP iterations and the fourth row the average time. The next four rows repeat these statistics for Integra_2.

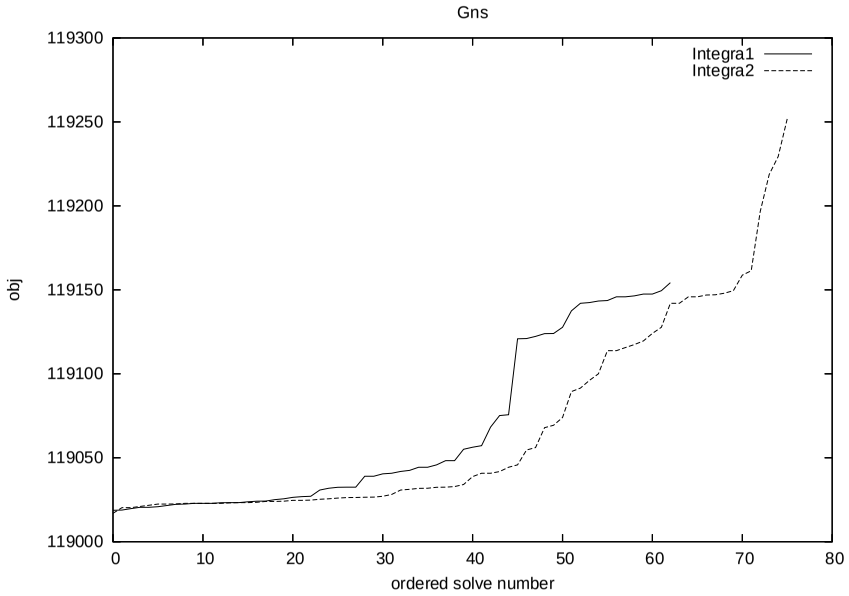
For number of successful solves, and for best objective value found, neither solver is better on all problems. The best objective value found by Integra_1 and Integra_2 was the same or similar for all problems. For average number of SLP iterations and average time Integra_2 is better on all problems. Note that the Slp5 code we introduced earlier is slower than both Integra_1 and Integra_2, as it takes an average of 5 seconds on Sonoco97 and 377 seconds on Sloten1relax (from Table 5.1.1).

We now look in more detail at the quality of all the different local minima found by Integra_1 and Integra_2. This is in response to concerns that since Integra_2 starts each solve with a large trust region, the effect of a random start point could be lost, and the same local minimum repeatedly found. For the problems Sonoco97, Sept98, Jan99 and Sept98, all of which have multiple local minima, we solved each problem 100 times by each code. For the four problems Integra_1 found what we believe to be the global minimum 7, 0, 2 and 1 times respectively, and Integra_2 found the suspected global minimum 6, 1, 4 and 4 times respectively. Figure 5.6 and Figure 5.7 plot all the objective values found by each code for each problem, ordered by increasing objective value. When comparing the Integra_1 and Integra_2 lines on one of the graphs, a lower line indicates that the solver found more points close to the global minimum. The leftmost entry of each line indicates the best objective value found, and how far each line stretches to the right indicates the number of successful solves.

The graphs show that for all four problems there are many local minima, all of which have similar objective values. Both solvers find many different local minima, and Integra_2 typically finds local minima at least as good as those found by Integra_1. Combined with the results in Table 5.2 these graphs indicate that Integra_2 is a faster solver than Integra_1, which is at least



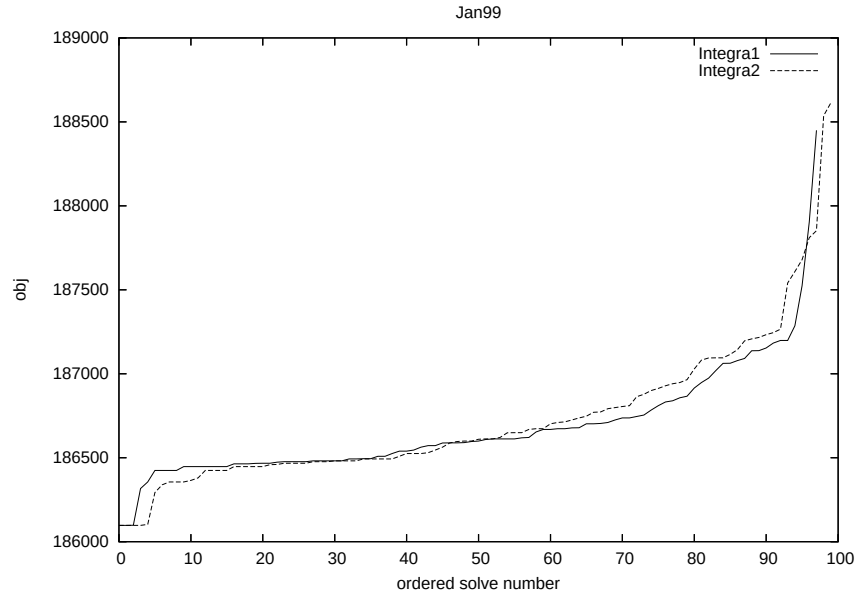
(a)



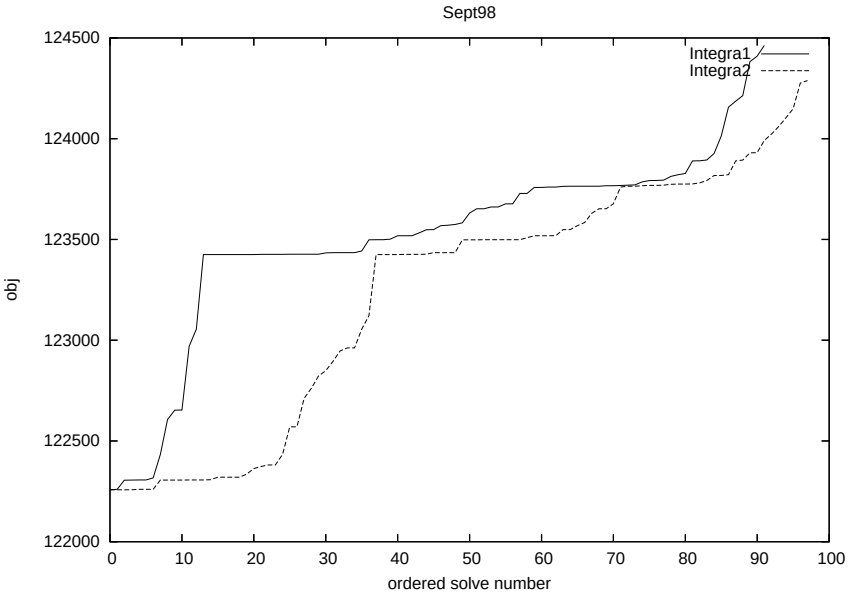
(b)

Figure 5.6: Comparison of ordered objective values for Sonoco97 (a) and Gns (b)

as good at finding the global minimum.



(a)



(b)

Figure 5.7: Comparison of ordered objective values for Jan99 (a) and Sept98 (b)

5.4 Stopping condition on number of solves

This section is about choosing an appropriate number of solves to do in one run of a problem, to be confident of having found the global minimum within some probability. Methods to actually increase the chance of finding the global minimum on any one solve are discussed in Section 6.3. Currently when solving GPPs Format International simply perform a fixed number of solves

of each problem. We consider other options here. The following analysis applies to the use of Integra_1 or Integra_2 as the solver.

We classify the different local minima to a problem into *classes*, based on their objective value. For problems which have very many local minima a class may consist of any local minimum whose objective value falls within a given range. The probability of any one solve finishing in a particular class we call the *class probability*, and the probability of any one solve finishing in the global minimum class we call the *global minimum class probability*.

The idea of classifying sample points from a population into classes is common in the literature, and applies to all sorts of different samples. Based on the sizes of the classes found in the sample the following three things can be estimated:

Case 1	The total number of classes in the population, which we call K.
Case 2	The probability that the relative sizes of the classes found in the sample is correct for the population.
Case 3	The probability that we have found the global minimum class, which we call $P(g)$.

Most of the literature concentrates on Case 1. Here we are interested in Case 3. To be confident of having found the global minimum we can do as many solves as are needed to make $P(g)$ sufficiently high. The following definitions will be useful to determine $P(g)$:

n	number of solves completed so far,
r	number of times the best class found so far has been found,
k	number of different classes found so far,
h_1	number of classes that have been found exactly once,
K	total number of classes,
α	global minimum class probability,
μ	mean of distribution of K,
v	variance of distribution of K,
E	parameter indicating how spread out the class probabilities are.

Note that n, r, k and h_1 are data that we generate during each run. K, α, μ, v and E are actual properties of each problem that may need to be estimated.

5.4.1 Literature review

Snyman and Fatti [1987] develop a criteria for when to stop doing new solves based on n and r. They make the key assumption that the global minimum class is as likely to be found as any other class. Although this assumption does not sound unreasonable it turns out to be quite strong. For example, if the same class is found in all of the first five solves, that class is thought

to almost certainly be the global minimum class. They also assume that α is a β-distribution, with known parameters a and b. Under these assumptions the probability of having found the global minimum class is:

$$P(g|n,r) = q_1(n,r) = \frac{\Gamma(n+a+b)\Gamma(2n-r+b)}{\Gamma(2n+a+b)\Gamma(n-r+b)}, \tag{5.17}$$

where Γ is the gamma function. Bolton et al. [2000] uses this formula with suitable, but unspecified, values of a and b. Each test problem is solved until $q_1(n,r)$ goes above a certain threshold. If this threshold is raised then more solves are performed and it is more likely the global minimum class is found.

Boender and Rinnooy Kan [1983] assume that all the class probabilities are equal. This leads to the probability of each new solve finding a new class of:

$$q_2(n,k) = \frac{k(k+1)}{n(n-1)} \qquad\qquad n \geq k+2. \tag{5.18}$$

Although this formula does not directly estimate $P(g)$, it can be used as a stopping rule by terminating a run once $q_2(n,k)$ is sufficiently small.

Good [1953] estimates the proportion of classes that are undiscovered, using h_1, the number of classes that have been found exactly once:

$$q_3(n,h_1) \simeq \frac{h_1}{n}. \tag{5.19}$$

This formula does not depend on the class probabilities being equal. It does not directly estimate $P(g)$, but can be used as a stopping rule by terminating a run once $q_3(n,h_1)$ is sufficiently small.

5.4.2 New method

Any of the test statistics $q_1(n,r)$, $q_2(n,k)$, and $q_3(n,h_1)$ can be used to give a stopping condition on a run of a problem. We now develop a new formula, which is similar to the Snyman Fatti formula $q_1(n,r)$, but depends no n and k instead of n and r. The formula has three simplifying assumptions, and ends up requiring the estimate of three priors.

$$q_4(n,k) = P(g|n,k) = \sum_K P(g|K,n,k)\ P(K|n,k), \tag{5.20}$$

$$= \sum_K P(g|K,n,k)\ \frac{P(k|K,n)\ P(K)}{P(k|n)}, \tag{5.21}$$

$$= \sum_K P(g|K,n,k)\ \frac{P(k|K,n)\ P(K)}{\sum_K P(k|K,n)\ P(K)}, \tag{5.22}$$

$$\simeq \sum_K \left(1-(1-\frac{1}{K})^n\right)\ \frac{P(k|K,n)\ P(K)}{\sum_K P(k|K,n)\ P(K)}, \tag{5.23}$$

$$\simeq \sum_K \left(1-(1-\frac{1}{K})^n\right)\ \frac{\mathrm{Po}(\lambda)\ P(K)}{\sum_K \mathrm{Po}(\lambda)\ P(K)}, \tag{5.24}$$

$$\simeq \sum_K \left(1-(1-\frac{1}{K})^n\right)\ \frac{\mathrm{Po}(\lambda)\ \mathrm{E}(K,\mu,v)}{\sum_K \mathrm{Po}(\lambda)\ \mathrm{E}(K,\mu,v)}. \tag{5.25}$$

Our formula is derived above. Throughout the derivation an expression like $P(g|K,n,k)$ means the probability of g given all of K, n and k. Also note that the sums over K are infinite sums over all possible values of K, and K could in theory could be any positive integer. However in practice the contribution to each sum from large values of K is minuscule, so in practice each summation is truncated after something like $K = 100$.

In (5.20) we rewrite $P(g|n,k)$ using the discrete form of the law of total probability, summing over all possible values of K. In (5.21) we rewrite $P(K|n,k)$ using Bayes theorem, and using the fact that $P(K|n) = P(K)$. In (5.22) we rewrite $P(k|n)$ using the discrete form of the law of total probability, again summing over all possible values of K and again using that $P(K|n) = P(K)$.

We now break down (5.22) further by considering each part, beginning with $P(g|K,n,k)$. We make the first simplifying assumption that if there are K classes the probability α of finding the global minimum on any one solve is $\frac{1}{K}$. Thus the basic probability formula $P(g|K,n) = 1-(1-\alpha)^n$ evaluates to $1-(1-\frac{1}{K})^n$. We make the second simplifying assumption that $P(g|K,n,k) = P(g|K,n)$, so $P(g|K,n,k) = 1-(1-\frac{1}{K})^n$ also. In (5.23) we therefore replace $P(g|K,n,k)$ with $1-(1-\frac{1}{K})^n$.

Next we consider $P(k|K,n)$. An exact form for $P(k|K,n)$ is given in Emigh [1983], under the assumption that the class probabilities are all known. For large K, if the class probabilities are equal, $K-k$ approximately follows a Poisson distribution (see e.g. Feller [1968]). If the class probabilities are unequal, $K-k$ can still be modelled as a Poisson distribution, but to account for the unequal class probabilities we must replace the sample size n with the *effective sample size*, denoted En, with $E \leq 1$. This leads to the estimate:

$$P(k|K,n) \simeq \mathrm{Po}(\lambda), \quad \lambda = (K-k)e^{-\frac{En}{K-k}}. \tag{5.26}$$

If the class probabilities are known to be equal then we use $E = 1$, else we must estimate a value for E less than 1. Thus in (5.24) we substitute $P(k|K, n)$ with $\text{Po}(\lambda)$, with λ as defined above. Note that λ depends on the index K, the problem data k and n, and the parameter E.

Finally we consider $P(K)$. This is the probability distribution of the total number of classes. Our third simplifying assumption is that this follows an Erlang distribution. This is not a strong assumption, as the Erlang distribution is quite general. The use of an Erlang distribution is in fact not crucial to the formula, and any other proper distribution could also be used. In (5.25) we substitute $P(k)$ with $E(K, \mu, v)$, where $E(K, \mu, v)$ is the Erlang distribution of K with mean μ and variance v.

We are now in a position to use the formula for $q_4(n, k)$ in (5.25). This formula has three simplifying assumptions, given above. $q_4(n, k)$ depends directly on data from each run, n and k, and also requires three prior estimates, μ, v and E. When solving large families of problems these prior estimates could be got from looking at the distribution of class sizes from previously solved problems. It is possible that in practice many of the GPPs solved will be very similar to previous GPPs, with the general structure of a family of problems determined by the layout of a factory, and only the raw costs, raw availability and product demands changing in different versions of a problem that occur over time. In this case the number of local minima is expected to be fairly constant for each solve, and it will be possible to get good estimates for the distribution of K. In this case $q_4(n, k)$ can be used to accurately predict the likelihood of having found the global minimum.

If the priors are not well estimated though, it is possible that the data from the run could be inconsistent with the prior estimates. For example, we may have a very small prior estimate of v, suggesting that the total number of classes is close to the estimated total μ, but find that $k \gg \mu$ so already many more classes than estimated have been found. In this case $q_4(n, k)$ may give misleading results.

Application of new method

Figure 5.8 plots $q_4(n, k)$ against k for different values of μ and v with $n = 20$. For these results we suppose that $E = 1$. Although the summation in (5.25) required to evaluate $q_4(n, k)$ is large enough to be impractical to evaluate by hand, it is more or less instant in a program like Maple.

There is a lot of information on the graph, and we now pick out some features. For all values of μ and v as k increases $q_4(n, k)$ decreases, because when many classes have been found it is less likely that the global minimum class has been found. When v is large k has a greater effect on $q_4(n, k)$ than when v is small, because when v is large the prior estimate for K is paid less heed than the actual number of classes found. When μ and v are both small $q_4(n, k)$ is high, because in this case we are very confident of there being a small number of classes, hence it is likely we have already found the global minimum class.

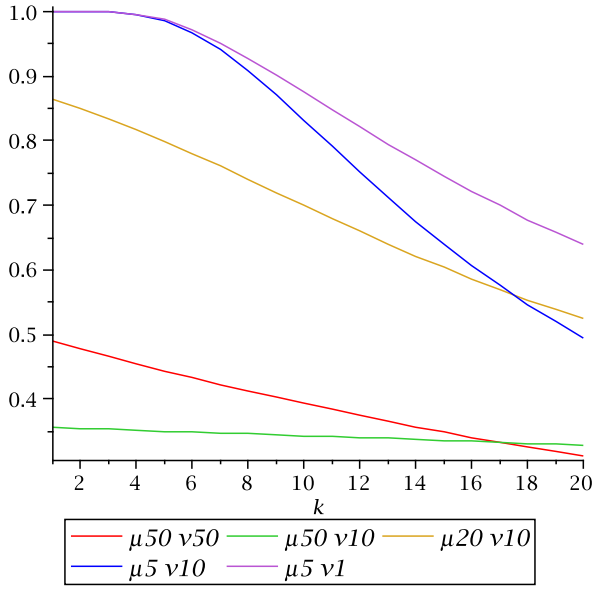


Figure 5.8: Probability of having found the global minimum, for different values of μ and v.

The wide range of probabilities in Figure 5.8 shows that $q_4(n, k)$ depends heavily on the priors μ and v. The test was repeated with $n = 50$. For all values of k, μ and v $q_4(n, k)$ was correspondingly higher than with $n = 20$, because once more solves are completed we are more confident of having found the global minimum.

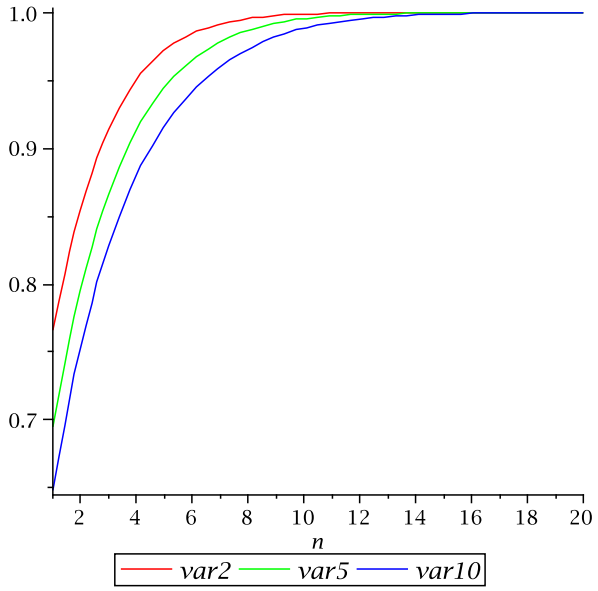


Figure 5.9: Probability of having found the global minimum

Figure 5.9 plots $q_4(n, 1)$ against n for a problem which is suspected of having only one class, and indeed only one was found on all trials. These problems are not uncommon in practice. Each of the three curves has expected number of classes $\mu = 1$ and a different variance. It can be seen that after 15 solves we are certain that we have the global minimum, even for the curve

with large variance.

Conclusion

We have shown there are several ways of calculating the probability of having found the global minimum. Which of $q_1(n,r), q_2(n,k), q_3(n,h_1)$ and $q_4(n,k)$ is best to use depends on how valid the assumptions that go into each formula are, and what prior estimates are available. $q_2(n,k)$ and $q_3(n,h_1)$ have the advantage of both being easy to calculate, whereas $q_1(n,r)$ and $q_4(n,k)$ require prior estimates of problem data.

Our formula $q_4(n,k)$ requires an estimate of the mean and variance of K, and the parameter E which indicates how varied the class probabilities are. We make simplifying assumptions about the probability of finding the global minimum and about the distribution of the total number of classes.

5.5 Summary

This chapter was about solving GPPs. We showed that SLP is an effective method for solving GPPs. For the GPP tight linear relaxations exist, by combining constraints from a Lagrangian-style relaxation with a linear programming relaxation. Solutions to the linear relaxations can be used as a warm start point for the solve of the original problem.

The solver Integra_1 is currently used by Format International to solve GPPs. Several improvements to this solver were suggested, affecting the preprocessing, trust region, ratio test, filter, phase one solve, LP solver and convergence. These improvements go into a new solver named Integra_2. Integra_2 solves the test problems more quickly than Integra_1, and is just as effective in finding the global minimum. Thus Integra_2 may be of commercial benefit to Format International.

We addressed the problem of how many random start solves to do in a run. After a review of the literature three existing methods were described, and a fourth method introduced and demonstrated. This method relies on a prior estimate for the distribution of the number of classes. This information could be found when solving many problems from the same family.

5.5.1 Further work

It is believed Format International are planning to adopt many of the changes suggested in Integra_2. As suggested above an interior point and SLP hybrid may be even better. It may also be of benefit to analyse whatever stopping rule Format currently use for determining the number of random starts to do, and perhaps suggesting an alternative rule.

Chapter 6

Multifactory Problem

In this chapter we introduce the multifactory problem. Several GPPs are combined to form what we call the *multifactory problem*, which is a large, but highly structured, NLP. This models the situation of one company who owns several factories, all of which are supplied from the same limited supply of raw materials. This is a new problem Format International envision solving in the future.

The problem is introduced in Section 6.1. In Section 6.2 we look at using decomposition to solve it. We show that by reformulation the problem can be decomposed into either a row linked or column linked problem. We attempt to solve it by Benders decomposition, the dual cutting plane method, and a modified augmented Lagrangian method. The augmented Lagrangian solve, with three innovations to the algorithm, is shown to take less estimated work than a direct solve of the undecomposed problem.

In Section 6.3 we introduce two new algorithms to find the global minimum of the multifactory problem, exploiting the fact that the constituent factories of a multifactory problem are nearly separable.

6.1 Introduction

Figure 6.1 shows an example multifactory problem named Ex1. Note that throughout this thesis example numbers are specific to the chapter they appear in, so the first example of each chapter is always Ex1. The problem consists of three factories. Each of these is a nonlinear general pooling problem, with mixing. The three factories are linked by a transport problem, which provides some of the raw materials for each factory from depots. The transport problem is a small linear problem where no mixing occurs, it simply sends the raw materials from the depots to the factories.

Collectively the transport problem and the factories are known as subproblems. In the multifactory problem we think of each factory as a general NLP, rather than specifically a

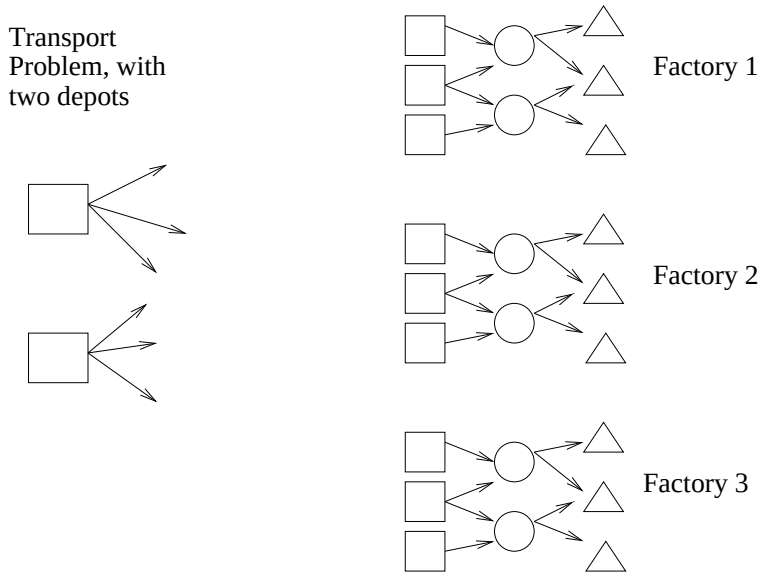


Figure 6.1: Multifactory problem Ex1

GPP. We label the transport problem with index 0 and the F factories with index $i \in 1 \dots F$. The variables of the transport problem are denoted $\boldsymbol{x^0}$, and are described further in Section 6.1.1. We use $\boldsymbol{x^i}$ for the general variables of factory i and $\boldsymbol{t^i}$ for the subset of $\boldsymbol{x^i}$ that are the raws that are used by several factories, known as the *shared raws*. We denote the (linear) objectives of the transport problem and factory i as f^0 and f^i respectively. We can then write the multifactory problem as the program $\mathcal{F}$ below:

$$\mathcal{F}: \quad f^* = \min_{\boldsymbol{x}} f(\boldsymbol{x}), \tag{6.1}$$

$$\text{s.t. } \boldsymbol{x} \in \boldsymbol{X}, \tag{6.2}$$

$$h(\boldsymbol{x}) = 0. \tag{6.3}$$

This problem may be referred to as the *primal problem*, to distinguish it from the *dual problems* that will come later. The objective, (6.1), can be separated into $f^0(\boldsymbol{x^0}) + \sum_{i=1}^{F} f^i(\boldsymbol{x^i})$, the sum of the objectives of the transport problem and the factories respectively. Constraints (6.2) can be separated into $\boldsymbol{x^0} \in \boldsymbol{X^0}$ and $\boldsymbol{x^i} \in \boldsymbol{X^i},\ i \in 1 \dots F$. These are the local constraints of the transport problem and the factories.

Constraint (6.3) is the linking constraint between the transport problem and the factories. It ensures that the total amount of each shared raw supplied by the transport problem to each factory is equal to the amount used by that factory. The exact form of the linking constraint depends on the model formulation, and is discussed in more detail later. The constraint can be written as $h(\boldsymbol{x^0}, \boldsymbol{t^1}, \dots, \boldsymbol{t^F}) = \boldsymbol{0}$ to highlight the contribution to the linking constraint from the transport problem and factories respectively. However this constraint cannot be rewritten as

separate constraints for the transport problem and the factories. Without the linking constraint the transport problem could be removed and the costs of supplying each shared raw absorbed into the factories. The problem would then decompose by factory. In some model formulations part of the linking constraints can be associated with each factory, in which case we refer to $\boldsymbol{v^i}$ as the linking constraint violation of factory i.

In the remainder of this section we will describe the transport problem more fully, describe the multifactory test problems to be solved, consider how the multifactory problem can be decomposed and look at the structure of possible subproblems resulting from decomposition.

6.1.1 Transport problem

F	number of factories, each indexed with superscript i
R	number of shared raws, each indexed with subscript j
D	number of depots, each indexed with subscript k
C^i_{jk}	cost of flow w^i_{jk}
w^i_{jk}	total mass flowing to factory i of shared raw j from depot k
y_{jk}	total mass of shared raw j leaving depot k
z^i_j	total mass that factory i is supplied of shared raw j

Table 6.1: Structural parameters and other parameters (all upper case) and variables (lower case) used in transport problem

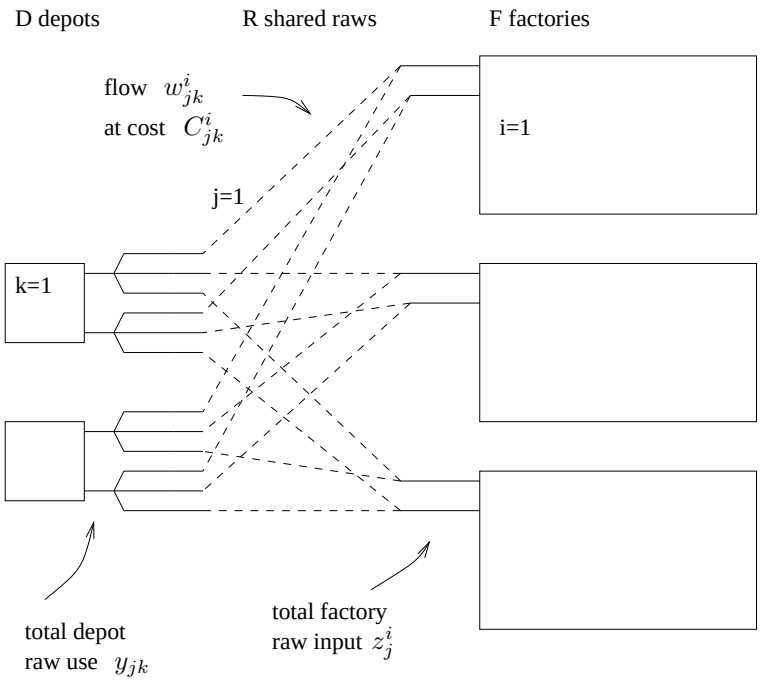


Figure 6.2: Ex1 transport problem and factories

Table 6.1 defines the parameters and variables used in the transport problem. Note that the transport problem is modelled with mass flow variables, not flow proportion variables as are used to model each factory. Figure 6.2 shows the Ex1 transport problem, which is a typical

transport problem. We can think of the shared raws flowing along arcs from the depots to the factories. Raws other than the shared raws have their costs absorbed into the factories and do not feature in the transport problem.

The transport problem is not a self contained problem, as it is linked to the factories by the linking constraints. If the multifactory problem is decomposed in the natural way (which we will later define precisely as decomposition with Split 1), the linking constraints are replaced by functions f_j^i, which reward supply of the shared raws to the factories. We can then give the full model of the transport problem after decomposition:

$$\min_{\boldsymbol{w},\boldsymbol{y},\boldsymbol{z}} \sum_{i=1}^{F}\sum_{j=1}^{R}\sum_{k=1}^{D} w_{jk}^i C_{jk}^i + \sum_{j=1}^{R}\sum_{k=1}^{D} f_{jk}(y_{jk}) + \sum_{i=1}^{F}\sum_{j=1}^{R} f_j^i(z_j^i) \tag{6.4}$$

$$\text{s.t.} \sum_{i=1}^{F} w_{jk}^i = y_{jk} \qquad j \in 1\ldots R,\ k \in 1\ldots D, \tag{6.5}$$

$$\sum_{k=1}^{D} w_{jk}^i = z_j^i \qquad i \in 1\ldots F,\ j \in 1\ldots R. \tag{6.6}$$

The objective, (6.4), contains three terms, each a large summation. The first is the total cost of flow along all arcs. The second is the sum of any cost terms applied to the use of the shared raws in the depots, and depends on how each function f_{jk} charges for y_{jk}, This depends on the type of transport problem, and is explained further in Section 6.1.2. The third is the sum of any cost terms applied to supply of shared raws to each factory, and depends on how each function f_j^i, charges for z_j^i, which may be varied during a decomposition algorithm. This is explained further in Section 6.1.4.

Constraint (6.5) defines the depot use $\boldsymbol{y}$ and constraint (6.6) defines the amounts supplied to the factories $\boldsymbol{z}$. As well as the constraints shown there are also non negativity bounds on flow, $\boldsymbol{w} \geq \boldsymbol{0}$, and simple upper bounds on shared raw use in depots, $\boldsymbol{y} \leq \overline{\boldsymbol{y}}$. All the constraints are linear hence the transport problem is linear whenever the objective is. The transport problem can always be decomposed by shared raw, though in practice the problem is easy enough to solve that decomposition is not necessary.

6.1.2 Test problems

We now introduce the multifactory problems that are solved in the later sections. These are all new problems, and consist of multiple factories linked by an appropriately sized transport problem. The two main multifactory problems are G and P. G consists of three small factories, each of which is a part of the decomposable factory Global_data (introduced in Section 4.2.5), linked by two shared raws. P consists of twelve large factories, each of which is similar to the Plant_1a factory (also introduced in Section 4.2.5), linked by four shared raws. These

multifactory problems were created as examples of the type of multifactory problems that could occur in real life.

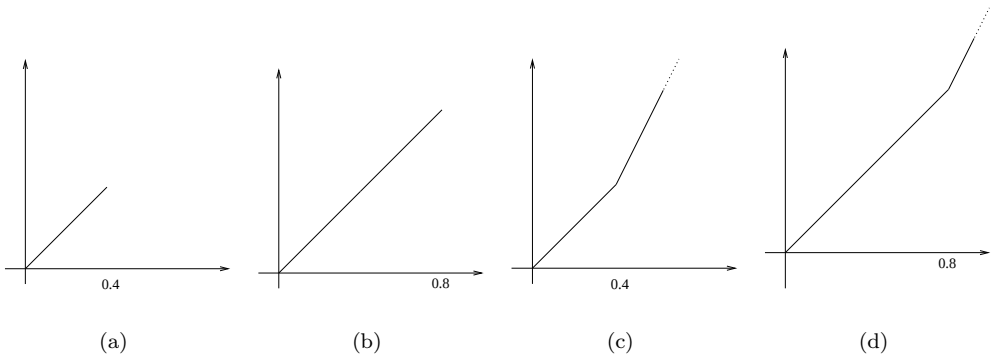


Figure 6.3: Cost of raw in transport problem (y-axis) against total amount of each raw stored in depots (x-axis), as a proportion of the amount desired by factories. For problem G (a), G_l (b), G_e (c) and G_{le} (d)

To form problem variants we vary the transport problem, either by varying $\overline{\boldsymbol{y}}$, the maximum mass of the shared raws available, or by varying the functions f_{jk} that determine the cost of shared raws in the depots. Figure 6.3 shows pictorially the costs on the raws the four different variants used.

In a standard problem (one with no subscript on the problem name, for example G or P_e) the total mass of each shared raw that can be supplied by the transport problem is about 40% of the total wanted by all the factories, after which no more is available. The price per unit of the raw is constant. In the problem variant G_l the amount of raw available is increased to 80%. In both G and G_l each function f_{jk} makes the price per unit supply of shared raw j from depot k constant. To form the problem variant G_e or P_e the limit on shared raw mass in each depot of the transport problem is replaced with a soft upper bound, so extra shared raw can be bought by the transport problem at an increased price. This is believed to better model the real life situation, and also has the effect of limiting the range of the dual value of the shared raws, which is of benefit when solving the multifactory problem with dual algorithms. To implement the soft upper bound we alter the function f_{jk} to give piecewise linear costs. The price per unit supply of shared raw j from depot k is constant up to the soft upper bound $\overline{y}_{jk}$, at which point it doubles. The further problem variant G_{le} combines the variants G_l and G_e by having a soft upper bound on the total shared raw mass of 80% of the total wanted by all the factories.

We call a transport problem with identical transportation costs on all arcs (for example no costs at all) a *simple transport problem*. A simple transport problem may be reformulated to have only one depot, as if all depots have similar characteristics we can coagulate them into one large depot. This can lead to a more effective decomposition approach. This reformulation is done in both the multifactory problems that have simple transport problems, called G_s and P_s.

We have so far described the features of problem G, G_l, G_e, G_s, P, P_e and P_s. The problems H_e and J_e are variants of G_e which have similar factory subproblems, but vary the number of shared raws or factories. Similarly Q_e and R_e are variants of P_e.

There are five further multifactory test problems, each made up of three different versions of a highly nonlinear factory, combined with a transport problem. The different factory versions differ in raw prices (all raws, not just shared raws), which simulates the local variations in price of raw materials. These five problems are named according to the factories that constitute them and the number of shared raws, so for example Sonoco97_1 is made up of versions of Sonoco97 linked by one shared raw.

problem name	F	R	D	approx total columns	approx total rows
G, G_l, G_e, G_{le}	3	2	3	1800	1900
G_s	3	2	1	1800	1900
H_e	3	4	3	1800	1900
J_e	6	2	3	3800	3700
P_e	12	4	4	45000	41900
P_s	12	4	1	45000	41900
Q_e	12	2	4	45000	41900
R_e	3	7	4	11600	10900
Sonoco97_1	3	1	3	3700	800
Sonoco97_5	3	5	3	3700	800
v1v_1	3	1	3	50	40
G1bConc_2	3	2	3	2200	2200
Sept98_5	3	5	3	2400	1400

Table 6.2: Multifactory test problems

Table 6.2 gives the number of factories, shared raws, depots in the transport problem and approximate columns and rows in the entire multifactory problem. A similar numerical scaling is applied to all problems.

6.1.3 Decomposition

In Section 6.2 we will solve multifactory problems by three different decomposition methods. There is one primal method, for which we must view the multifactory problem as column-linked, and two dual methods, for which we must view the multifactory problem as row-linked. Figure 6.4 demonstrates two ways of splitting the multifactory problem. Both of these can each lead to either a column-linked or a row-linked form.

Split 1

Split 1 is the natural way to decompose the multifactory problem. The split comes just before the factories, resulting in a relatively large transport problem, which was seen in model (6.4)-(6.6) given above. With this split what ties together the transport problem and the factories is that the mass of each shared raw supplied to a factory must be equal to the mass of the

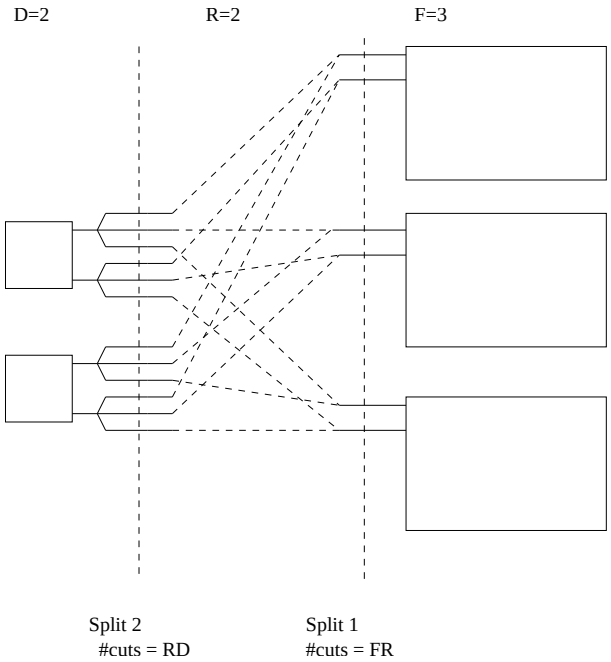


Figure 6.4: Two ways to decompose Ex1

shared raw used by that factory. This gives the linking constraints below, which are added to the existing transport model:

$$z_j^i = t_j^i \qquad\qquad i \in 1 \ldots F,\ j \in 1 \ldots R, \tag{6.7}$$

where z_j^i is mass to factory i of raw j supplied, and t_j^i is the mass that factory i uses of raw j. Note that these linking constraints are separable by factory.

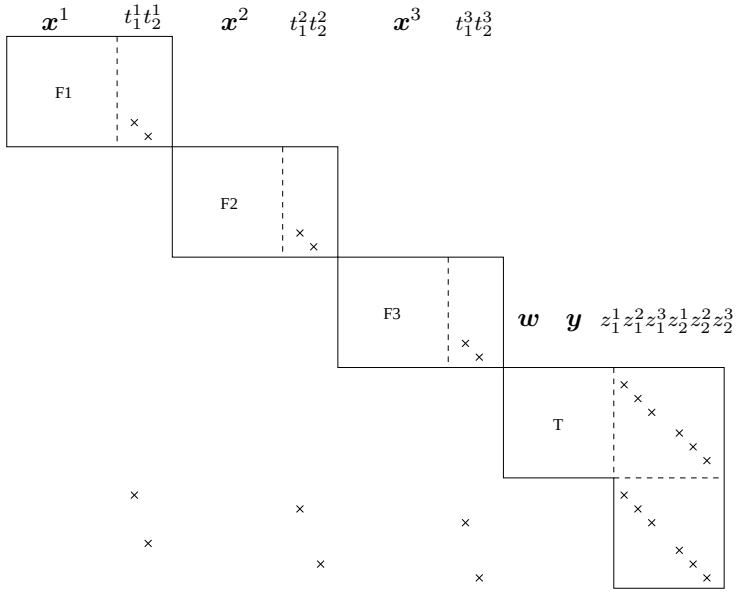


Figure 6.5: Constraint matrix of Ex1 with Split 1

Figure 6.5 shows the structure of the constraint matrix of Ex1 after decomposition by Split 1. There are three factories and a transport problem. For presentation purposes we separate each subproblem into different parts, to draw attention to some of the variables. For example for the first factory, $F1$, we have the vector of variables $\boldsymbol{x^1}$ then the shared raw usage variables t_1^1 and t_2^1. For the transport problem we have the vector variables for flow and depot use, $\boldsymbol{w}$ and $\boldsymbol{y}$ respectively, then the individual $\boldsymbol{z}$ variables for supply to each factory. The problem has been slightly reformulated with duplication of $\boldsymbol{z}$ into new rows in the transport problem. This is an example of *goal coordination* (see e.g. Wagner [1997]), where the problem dimensionality is slightly increased so that the linking constraints are simple equalities. By considering Figure 6.5 we see that the problem may be seen as column linked, with FR linking columns, t_j^i $i = 1 \ldots F,\ j = 1 \ldots R$, enforcing (6.7). It may also be seen as row linked, with the FR linking rows , z_j^i $i = 1 \ldots F,\ j = 1 \ldots R$ across the bottom of the figure enforcing (6.7).

Split 2

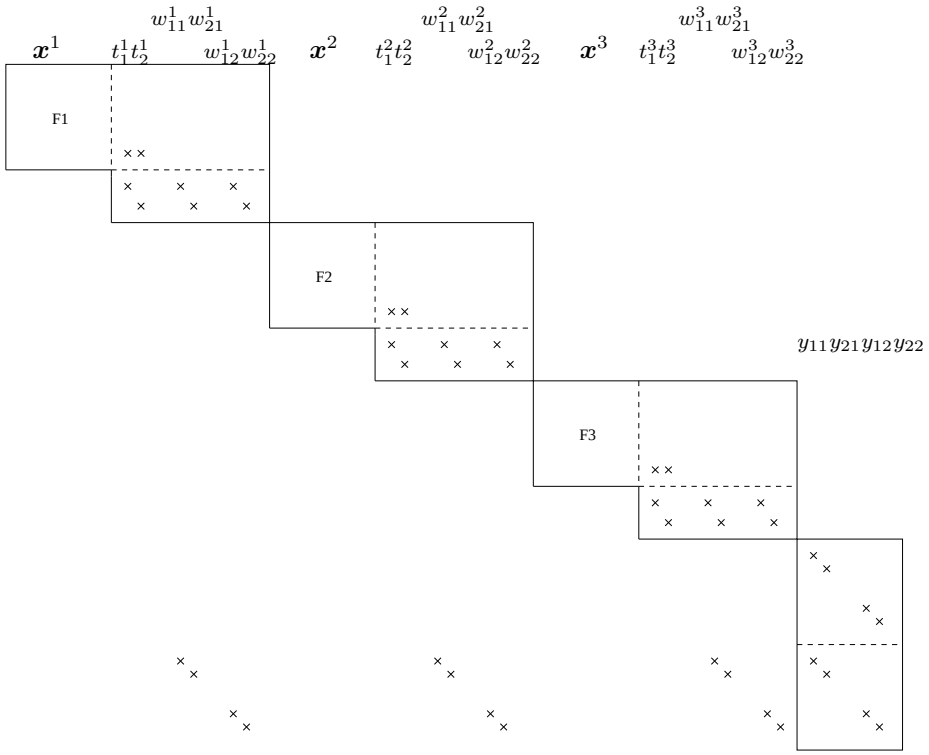


Figure 6.6: Constraint matrix of Ex1 with Split 2

Split 2 also breaks the multifactory problem into transport and factory subproblems, but results in a small transport problem, that only contains the $\boldsymbol{y}$ variables (as can be seen from the position of the split in Figure 6.4). The $\boldsymbol{z}$ variables are not needed at all in this formulation and the shared raw flow variables $\boldsymbol{w}$ are now included in each factory instead of the transport problem. Figure 6.6 shows the structure of the constraint matrix of Ex1 decomposed with Split 2. There is a reformulation with $\boldsymbol{y}$ duplicated into new rows in the transport problem.

Using this split the multifactory problem may be viewed as having FRD linking columns, w^i_{jk} $i = 1 \dots F,\ j = 1 \dots R,\ k = 1 \dots D$, or RD linking rows, y_{jk} $j = 1 \dots R,\ k = 1 \dots D$. For problems with few depots compared to factories RD linking rows is fewer than the FR linking rows of Split 1. In particular, for problems with a simple transport problem where all depots are equivalent the transport problem can be reformulated to have only one depot so the formulation shown in Figure 6.6 can be simplified to a multifactory problem with only R linking rows. In this case the linking constraints can be written as:

$$y_j = \sum_{i=1}^{F} t^i_j \qquad\qquad j \in 1 \dots R. \tag{6.8}$$

Note that these linking constraints are also separable by factory. When solving most multifactory problems we use Split 1, and solve either a column linked problem with FR linking columns or a row linked form with FR linking rows. For multifactory problems with a simple transport problem we use Split 2, resulting in a row linked problem with D linking rows.

6.1.4 Subproblems

Decomposition by either Split 1 or Split 2 results in a (linear) transport problem and several (nonlinear) factories. During a solve the objective to all these subproblems may be altered. Recall that $\boldsymbol{t}$ is the subset of the $\boldsymbol{x}$ variable denoting the mass of the shared raws. The general form of a subproblem, omitting the factory superscript i for clarity, is given below:

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T\boldsymbol{t} + \boldsymbol{f}_t(\boldsymbol{t}) \\ &\text{s.t. } \boldsymbol{x} \in \boldsymbol{X}, \end{aligned} \tag{6.9}$$

where $\boldsymbol{c}$ and $\boldsymbol{\lambda}$ are appropriately sized constant vectors, $\boldsymbol{f}_t$ is a general cost function, $\boldsymbol{x} \in \boldsymbol{X}$ are general constraints on $\boldsymbol{x}$. $\boldsymbol{\lambda}$ will later be identified as the vector of dual variables on the linking constraints. In the objective function the first term, $\boldsymbol{c}^T\boldsymbol{x}$, is the (linear) private objective of the subproblem. The second term, $\boldsymbol{\lambda}^T\boldsymbol{t}$, (linearly) penalises or rewards use of shared raws at rate $\boldsymbol{\lambda}$. The third term, $\boldsymbol{f}_t(\boldsymbol{t})$, is a general function of the use of shared raws. For the subproblem that is the transport problem, with objective shown in (6.4), the use of the shared raws is denoted by $\boldsymbol{z}$, so $\boldsymbol{f}_t$ can thought of as corresponding to f^i_j.

Together $\boldsymbol{\lambda}$ and $\boldsymbol{f}_t$ control how use of the shared raws affects the cost of a subproblem. Their nature is determined by the decomposition algorithm used. Figure 6.7 shows three ways that $\boldsymbol{\lambda}$ and $\boldsymbol{f}_t$ can affect the cost function, for a problem with one shared raw. With Pen 1 $\boldsymbol{\lambda}$ is nonzero and there is no $\boldsymbol{f}_t$ term. $\boldsymbol{\lambda}$ alters the gradient of the raw mass response curve so the

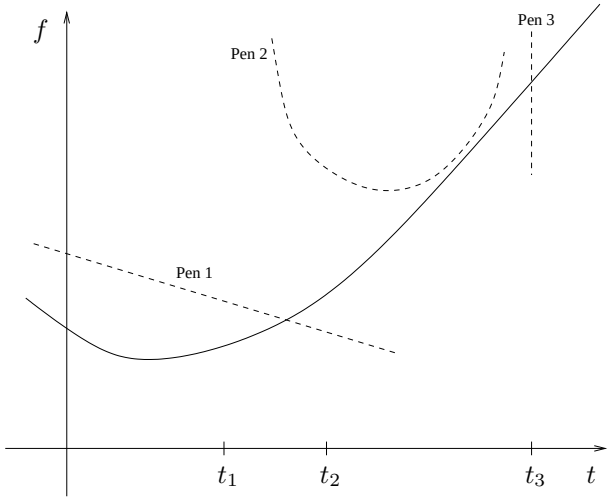


Figure 6.7: Raw mass response curve for a factory (solid line representing $\boldsymbol{c}^T\boldsymbol{x}$) with three examples of penalties that could be added to the objective (dashed lines representing $\boldsymbol{\lambda}^T\boldsymbol{t} + \boldsymbol{f}_t(\boldsymbol{t})$).

minimum value of t is at t_1. With Pen 2 $\boldsymbol{\lambda} = \boldsymbol{0}$ and $\boldsymbol{f}_t$ is a convex penalty on deviation of t from some target t'. With this penalty the optimal value of t is at t_2. With Pen 3 $\boldsymbol{\lambda} = \boldsymbol{0}$ and $\boldsymbol{f}_t$ is an infinite penalty forcing the solution $t = t_3$. In the following sections we will see subproblems with each of Pen 1, Pen 2 and Pen3, as well as with both Pen 1 and Pen 2 together.

6.2 Solving by decomposition

In this section we introduce three solve methods for the multifactory problem, all using decomposition. Although the solve methods are not unusual, they have not previously been applied to the multifactory problem. The methods all assume to some extent that the subproblems are convex, which is highly unlikely in practice. When applied to nonconvex problems there is no guarantee of finding the global minimum, or even a local minimum. However, when applied to problems that are only slightly nonlinear, such as G and P, the methods still give an excellent chance of finding the global minimum in any one solve. In Section 6.3 we look at specific methods for finding the global minimum of multifactory problems.

We have seen above that the multifactory problem may be viewed as either row or column linked. This means that there are many possible approaches to solve the problem. Using the column linked form Benders decomposition is possible. Using the row linked form we may use dual cutting plane or augmented Lagrangian. These three solve methods are explored in the sections below.

6.2.1 Benders decomposition

Benders decomposition is one of a class of primal methods that can be applied to column linked problems. An outline of Benders decomposition is given in Section 3.2. Benders decomposition iteratively fixes the values of the linking variables to give separable subproblems, which is like applying a penalty of type Pen 3 to each subproblem.

We applied Benders decomposition to the multifactory problem decomposed with Split 1, which can be viewed as a column linked problem with FR linking columns. Note that here after relaxing the linking columns the subproblems are still NLPs, so Benders decomposition may fail. In each outer iteration of our algorithm all the cuts from each subproblem were added together to form an aggregated cut, which was then added directly to the transport problem. Thus the transport problem, with the addition of cuts from the factories, played the role of the master problem.

	G	G_l	G_e	G_{le}	H	J	G_s	P	P_s
Benders	13		17					fail	

We tested the algorithm on two small and one large problem. G and G_e are almost linear problems so Benders decomposition was able to solve them well, in 13 and 17 outer iterations respectively, where one outer iteration is a solve of all factories and the enhanced transport problem. P is a much larger and more nonlinear problem and did not solve well with Benders decomposition. It failed after 67 iterations due to the nonconvexity of the problem producing conflicting feasibility cuts in the master problem.

6.2.2 Lagrangian relaxation

In section 3.3 we introduced Lagrangian relaxation as one of the ways of solving row linked problems. Lagrangian relaxation removes the linking rows from the problem and places them in the objective, with costs on them. The resulting problem decomposes into subproblems that can then be solved to update the costs on the linking rows. There are different ways of updating the costs. In Section 6.2.3 we consider updating them by the dual cutting plane method. In Section 6.2.4 and 6.2.5 we use augmented Lagrangian.

The Lagrangian program $\mathcal{L}(\boldsymbol{\lambda})$ is formed from the primal program $\mathcal{F}$ by relaxing the linking constraint $h(\boldsymbol{x}) = \mathbf{0}$ from (6.3) and placing it in the objective with penalty $\boldsymbol{\lambda}$.

$$\mathcal{L}(\boldsymbol{\lambda}): \quad l^*(\boldsymbol{\lambda}) = \min_{\boldsymbol{x}} l(\boldsymbol{x}, \boldsymbol{\lambda}) = \ f(\boldsymbol{x}) + \boldsymbol{\lambda} h(\boldsymbol{x}), \qquad (6.10)$$
$$\text{s.t. } \boldsymbol{x} \in \boldsymbol{X}.$$

We refer to the surface caused by varying $\boldsymbol{\lambda}$ in $l^*(\boldsymbol{\lambda})$ as the dual response curve. As the linking

constraints are separable by factory the problem is now decomposed into separate factories and the transport problem. These are known as the subproblems. The dual program of $\mathcal{L}(\boldsymbol{\lambda})$ is:

$$\mathcal{L}_D : \quad l_D^* = \max_{\boldsymbol{\lambda}} \; l^*(\boldsymbol{\lambda}). \tag{6.11}$$

Theorem 5. $l_D^* \leq f^*$.

where, l_D^* is the dual objective and f^* is the global minimum of the primal problem $\mathcal{F}$. From Lemarechal [2001] page 114. The theorem says that the dual objective value is always an underestimate of the primal objective value. If $l_D^* < f^*$ it is said there is a *duality gap*. If $l_D^* = f^*$ then there is no duality gap and we may find the optimal objective value of $\mathcal{F}$ by solving $\mathcal{L}_D$.

6.2.3 Dual cutting plane method

In the dual cutting plane method we solve $\mathcal{L}_D$ by finding the $\boldsymbol{\lambda}$ that maximise the dual response curve. The dual cutting plane method builds up cuts from the subproblems and places them in a master problem, in a similar way to Benders decomposition. An outline of the method is given in the Table below.

1	Fix the costs $\boldsymbol{\lambda}$ on the linking rows.
2	Solve the resulting series of smaller NLPs.
3	Add a cut to the master problem.
4	Solve the master problem, to find new costs for the linking rows.

Table 6.3: An outer iteration of the dual cutting plane method

The aim each iteration is to solve (6.10). In Step 1 we fix $\boldsymbol{\lambda}$ and in Step 2 solve all the subproblems. Each subproblem i gives a dual objective $l^i(\boldsymbol{\lambda})$ and use of shared raw j of t_j^i. Fixing $\boldsymbol{\lambda}$ like this can be thought of as applying a penalty of type Pen 1 to each subproblem. In Step 3 summing the objective value and use of shared raws from all the subproblems provides a linear cut which overestimates the dual response curve. The linear cuts from all iterations form the *master problem*. The master problem after K iterations is shown below.

$$\min_{\boldsymbol{\lambda},z} \; z, \tag{6.12}$$

$$\text{s.t.} z \leq l(\hat{\boldsymbol{\lambda}}_k) + (\boldsymbol{\lambda} - \hat{\boldsymbol{\lambda}}_k)\hat{\boldsymbol{h}_k} \qquad k = 1 \ldots K, \tag{6.13}$$

$$\boldsymbol{\lambda} \in \Lambda. \tag{6.14}$$

We introduce a variable z to represent the objective value. The objective (6.12) is to maximize z. The constraints (6.13) limit z to be below all the linear constraints that have been generated

by each cut. $\hat{\boldsymbol{\lambda}}_k$ is the value of $\boldsymbol{\lambda}$ at iteration k, and $\hat{\boldsymbol{h}}_k$ the value of the constraint violation at iteration k. When $\boldsymbol{\lambda}$ has dimension greater than one the constraints of (6.13) will be planes, hence the name dual cutting plane method. The constraints (6.14) are optional constraints on $\boldsymbol{\lambda}$ which are discussed further in the section below on the master problem.

In Step 4 maximising the master problem gives a new value of $\boldsymbol{\lambda}$ to be used in the next iteration, and the process repeats. Eventually the dual response curve is well enough approximated to find l_D^* to given accuracy.

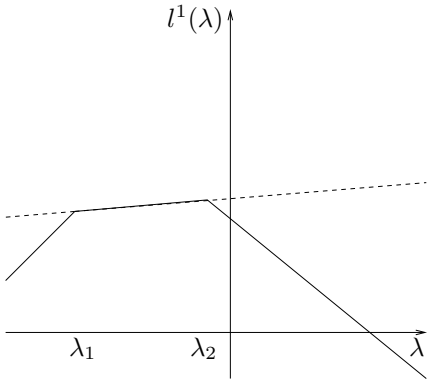


Figure 6.8: Example factory Ex2

Figure 6.8 shows the dual response curve (solid line) approximated by one linear cut (dashed line) for the small linear factory Ex2. This problem has just one shared raw. We think of Ex2 as being the first factory subproblem of a multifactory problem.

Theorem 6. *$l^*(\boldsymbol{\lambda})$ is concave, regardless of whether or not f is convex.*

This is Theorem 10 in Lemarechal [2001]. The theorem implies that the linear cuts that approximate the dual response curve in the master problem always give upper bounds on it. Furthermore, as the dual response curve is concave, any cuts from suboptimal points (those with objectives above $l^*(\boldsymbol{\lambda})$) will still be valid, though not tight. This leads to the corollary below:

Theorem 7. *Any feasible (but not necessarily optimal) solution of the subproblems gives a cut that is an upper bound on $l^*(\boldsymbol{\lambda})$.*

This theorem is important as it says that even when the subproblems are not solved to global optimality a valid cut is still generated, which may be added to the master problem. This gives an advantage to a dual method such as this to a primal method like Benders decomposition. In Benders decomposition it is possible that the subproblems do not provide valid cuts, even if they are solved to global optimality. This is believed to be the reason that Benders decomposition could not solve the large GPP in Chapter 3.

Master problem

In each outer iteration of the dual cutting plane method the master problem is solved to find a new value of $\boldsymbol{\lambda}$. The master problem is a simple linear maximisation problem. However the values of the $\boldsymbol{\lambda}$ variables suggested from one solve can be very different to the $\boldsymbol{\lambda}$ from the iteration before. This can lead to slow convergence and many outer iterations. Because of this it is common to modify the master problem to stabilise movement in $\boldsymbol{\lambda}$ from iteration to iteration. This modification can also have the effect of preventing the master problem from being unbounded, which can be a problem on early iterations when there are few cuts in the master problem. It can also make warm starting subproblems more effective, as with little movement in $\boldsymbol{\lambda}$ each subproblem is similar to the previous one solved. We give some of the methods to stabilise $\boldsymbol{\lambda}$ below.

One method is to take a convex combination of the suggested $\boldsymbol{\lambda}$ from the most recent iteration with the $\boldsymbol{\lambda}$ from the iteration that has lead to the best solution values so far (see e.g. Wentges [1997]). Another method is to make $\boldsymbol{\lambda}$ the analytic center of all the cuts so far that are considered good cuts (see e.g. Boyd and Vandenberghe [2003]). Agarwal et al. [1989] uses simple bounds on $\boldsymbol{\lambda}$ to limit it to be within a range that is anticipated to contain the optimal solution. In trust region methods these simple bounds can be moved if they become active, so are guaranteed to contain the optimal $\boldsymbol{\lambda}$. In some algorithms if a measure of progress is met the trust region can be increased. See e.g. Marsten [1975], Kallehauge et al. [2001] or Lim and Sherali [2006]. Ben-Amor and Desrosiers [2006] uses *stabilised column generation*, which when applied to the dual stabilises the $\boldsymbol{\lambda}$ variables. In this method a quadratic penalty (or any other convex penalty) is added to the master problem to penalise movement in $\boldsymbol{\lambda}$ away from the value at the previous iteration (see also Lemarechal [2001], Frangioni [2002]).

Modifying the master problem to stabilise $\boldsymbol{\lambda}$ can be done using the general $\boldsymbol{\lambda} \in \Lambda$ constraint of (6.14), and by modifying the objective. In what we call algorithm ACP1 we solve multifactory problems using the dual cutting plane method, and stabilise $\boldsymbol{\lambda}$ with a fixed size trust region. This is a constraint of the form $\boldsymbol{\lambda} - \hat{\boldsymbol{\lambda}} \leq \boldsymbol{\rho}$, where $\hat{\boldsymbol{\lambda}}$ is the old value of $\boldsymbol{\lambda}$ and $\boldsymbol{\rho}$ is a vector of constant entries giving the trust region size. The stabilisation of ACP1 does not introduce any curvature, so at the solution it is likely the solution found will not be primal feasible. However, we will see that in most cases it is easy to rectify this.

In algorithm ACP2 we stabilise $\boldsymbol{\lambda}$ using a fixed size trust region and a quadratic penalty. Constraint (6.14) becomes $\boldsymbol{\lambda} - \hat{\boldsymbol{\lambda}} \leq \boldsymbol{\rho}$ and the objective (6.12) becomes $\min_{\boldsymbol{\lambda}, z} z + \boldsymbol{\epsilon}(\boldsymbol{\lambda} - \hat{\boldsymbol{\lambda}})^2$ where $\boldsymbol{\epsilon}$ is a vector of constant entries giving the amount of curvature. Because of this curvature ACP2 is usually able to find a primal feasible point at the solution. Note that in ACP2 the trust region is retained, as even with the quadratic penalty present having a trust region was found to save iterations.

Primal and dual response curves

The primal violation response curve for subproblem i shows how the objective value, $f^i(x)$, varies with the violation of the linking constraints, $\boldsymbol{v}$. The dual response curve, introduced above, shows how the dual solution, $l^i(\boldsymbol{\lambda})$ varies with the dual variable, $\boldsymbol{\lambda}$.

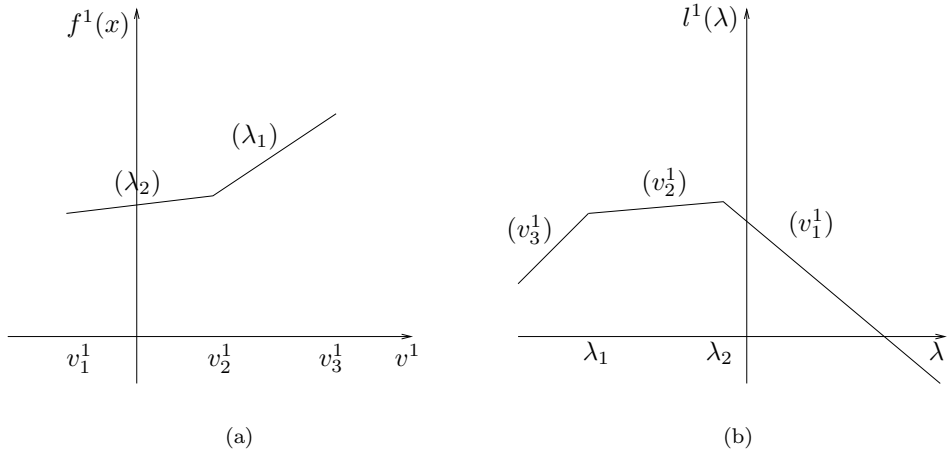


Figure 6.9: Primal violation response curve (a) and dual response curve (b) for example factory Ex2

Figure 6.9 shows the primal violation response curve and dual response curve for an example factory Ex2, which has one shared raw. Symbols in brackets are gradients. The position of the break points in the primal graph correspond to the gradients in the dual graph, and vice versa. Such graphs are called *conjugate graphs*.

The primal graph is piecewise linear if the problem is an LP and convex if the problem is convex. At the break points of the primal graph $\boldsymbol{\lambda}$ is degenerate. The feasible point, $v^1 = 0$, corresponds to a unique $\boldsymbol{\lambda}$.

The dual graph is linear if the primal problem is linear, and is always concave, by Theorem 6. At the break points of the dual graph v^1 is degenerate. The solution of the dual problem is the highest point on the the dual response curve, which usually occurs at a break point. This is the case in Figure 6.9(a), where the solution is at $\boldsymbol{\lambda}_2$. Therefore whenever the apex of the dual graph is linear, as can commonly happen even on nonlinear problems, the primal problem is degenerate.

Figure 6.10 shows the relationship between $\boldsymbol{\lambda}$ and the constraint violation v^1 in Ex2. This graph can be deduced from Figure 6.9(b). For different values of $\boldsymbol{\lambda}$ there is a different amount of primal violation in the primal problem. In Figure 6.10 the optimal $\boldsymbol{\lambda}$ is the one that allows $v^1 = 0$, at which point v^1 is again degenerate.

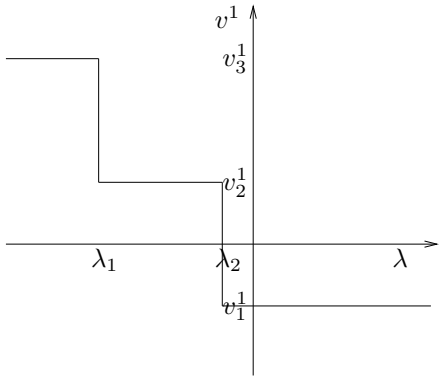


Figure 6.10: (λ, v^1) graph for the Lagrangian formulation of Ex2

Primal degeneracy

When looking at the primal and dual response curves above we saw that the primal problem can be degenerate with the optimal $\boldsymbol{\lambda}$. In this case it is almost certain that the primal solution found will not satisfy the linking rows. This is a weakness of the dual cutting plane method. As there is a large amount of linearity in the pooling problems primal degeneracy is expected to occur often when solving with a standard dual cutting plane method, like ACP1. However, in methods that a quadratic term to the objective, such as ACP2, it is possible that the dual response curve will be strictly concave at the solution, and so degeneracy will be avoided.

Lemarechal [2001] shows that if there is no duality gap (and certain other regularity conditions are met) the primal feasible solution $\boldsymbol{x^*}$ can be extracted directly from the dual solution after solving with dual cutting plane. If there is no duality gap it is also possible to recover the primal feasible solution by simply adjusting some primal values in each of the subproblems that has been made degenerate by the optimal $\boldsymbol{\lambda}$.

A third method for recovering primal feasibility is to add a quadratic (or other strictly convex) penalty to each subproblem, penalising the contribution to constraint violation from that subproblem. With the optimal $\boldsymbol{\lambda}$ the subproblems are likely to be degenerate, with many solutions with equal objective value, and a quadratic penalty can gently nudge the primal values towards feasibility without worsening the objective value. The quadratic penalty can in fact be added from the start of the dual cutting plane method, and included in each solve of the subproblems while we are still finding the optimal $\boldsymbol{\lambda}$. This is done by adding the term $\epsilon(\boldsymbol{h^i})^2$ to the objective of subproblem i, where ϵ is a constant parameter for the size of the penalty, known as the curvature. With the existing $\boldsymbol{\lambda}$ penalties and the quadratic penalty each subproblem now has penalties of both type Pen 1 and Pen 2.

Although we have approached it from another angle, such a quadratic penalty is identical to a quadratic penalty added to the master problem to stabilise $\boldsymbol{\lambda}$, such as in our algorithm ACP2. With this term when ϵ is large the cuts generated at each iteration can be different to those that would have been generated from solving with no quadratic penalty. In ACP2 we

therefore calculate for each cut what $\boldsymbol{\lambda}$ would have given the resulting $l^*(\boldsymbol{\lambda})$ and $\boldsymbol{v}$ without a quadratic penalty, and add the resulting cut to the master problem.

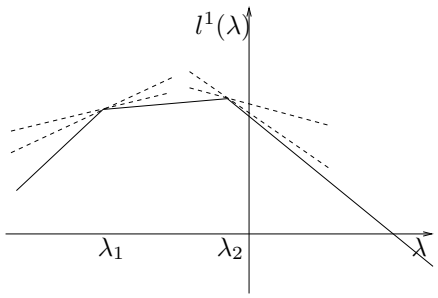


Figure 6.11: Dual response curve to factory Ex2 (solid line), with example cuts generated by ACP2 (dashed lines)

Figure 6.11 shows some of the cuts that could be generated when solving Ex2 using ACP2 with a large value of ϵ. The cuts are tangential to the dual response curve, so while valid are not tight. With an algorithm like ACP1 the cuts will all be tight on the dual response curve, as in Figure 6.8. The advantage of being able to produce all the cuts of Figure 6.11 is that the dual response curve is no longer degenerate with the optimal $\boldsymbol{\lambda}$, so there is no primal degeneracy. The disadvantage, as we will see below, is that because the cuts are not all tight more cuts are generally needed, so extra iterations are taken.

Results

As the dual cutting plane method is a row linked method we will therefore apply it to the row linked form of Split 1 for most problems, using the row linked form of Split 2 for problems with a simple transport problem. These problems have FR and R linking rows respectively.

	G	G_l	G_e	G_{le}	H	J	G_s	P	P_s
$\lvert\boldsymbol{\lambda}\rvert$	6	6	6	6	12	12	2	48	4
ACP1	34	25	27	43	69	64	8	>500	25
ACP2	43	28	34	41	86	91	14		

Table 6.4: Results for dual cutting plane method

Table 6.4 gives the number of outer iterations to solve test problems using either algorithm ACP1 or ACP2. Note that ACP1 does not find a primal feasible solution, though as stated above it is easy to find a feasible point from the dual solution when there is no duality gap.

The first row of the table gives the number of $\boldsymbol{\lambda}$ variables, which is equivalent to the number of shared raws and to the dimension of the master problem. The second and third rows give the number of outer iterations to solve the problems. Problems with many $\boldsymbol{\lambda}$ variables took significantly longer to solve with both algorithms. This is clear when comparing results for G with G_s, and especially when comparing results for P with P_s. ACP1 consistently takes fewer outer iterations than ACP2, and in fact also takes fewer total SLP and LP iterations (not

shown). In ACP2 with the quadratic penalty the subproblems generate less cuts that are tight in the master problem, so more iterations are needed to find the correct $\boldsymbol{\lambda}$.

The dual cutting plane method (either ACP1 or ACP2) is slower than Benders decomposition on the problems that Benders could solve(G and G_e). However the dual cutting plane method was able to solve many more problems, and ACP1 was very effective in particular at solving problems with few linking variables, namely G_s and P_s.

6.2.4 Augmented Lagrangian introduction

We now look at solving the multifactory problem by the augmented Lagrangian method. In this section we introduce the method. In Section 6.2.5 we describe its application to the multifactory problem. Like the dual cutting plane method the augmented Lagrangian method is a dual method for solving row linked problems. It was introduced independently by Hestenes [1969] and Powell [1969].

$$\mathcal{W}(\boldsymbol{\lambda}, \epsilon): \quad w^*(\boldsymbol{\lambda}, \epsilon) = \min_{\boldsymbol{x}} w(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon) = f(\boldsymbol{x}) + \boldsymbol{\lambda} h(\boldsymbol{x}) + \epsilon h(\boldsymbol{x})^2 \tag{6.15}$$
$$\text{s.t. } \boldsymbol{x} \in \boldsymbol{X}.$$

Above we given the augmented Lagrangian problem derived from relaxing the linking equality constraints of $\mathcal{F}$ of (6.3) and placing them in the objective with a linear penalty weighted by $\boldsymbol{\lambda}$, and a quadratic penalty weighted by ϵ. The resulting objective, $w(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon)$, is called the *standard augmented Lagrangian function*. When the linking constraints are instead inequalities, $g(\boldsymbol{x}) \leq \mathbf{0}$, they must be converted to equality constraints by the addition of an extra variable. This leads to the resulting augmented Lagrangian function, given in e.g. Rockafellar [1974] and Fletcher [1981]:

$$w_1(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon) = f(\boldsymbol{x}) + \boldsymbol{\lambda} \max(g(\boldsymbol{x}), -\frac{\boldsymbol{\lambda}}{2\epsilon}) + \epsilon \max(g(\boldsymbol{x}), -\frac{\boldsymbol{\lambda}}{2\epsilon})^2. \tag{6.16}$$

Sun et al. [2005] give four different types of augmented Lagrangians for inequality constraints. These are the essentially quadratic ($w_1(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon)$, shown above), the exponential-type, the modified barrier and the penalised exponential-type. Rockafellar and Wets [1998] define a general augmenting function as any well behaved function that has zero value when constraint violation is zero and has greater values than this elsewhere. Huang and Yang [2003] show that the general augmenting function has no duality gap.

From now on we consider only the standard augmented Lagrangian function $w(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon)$ from (6.15). With this objective after decomposition all the subproblems have linear and quadratic penalties, so the augmented Lagrangian method can be thought of as applying a penalty of

type Pen 1 and Pen 2 to each subproblem.

The augmented Lagrangian dual program is:

$$\mathcal{W}_D(\epsilon): \quad w_D^* = \max_{\boldsymbol{\lambda}} w^*(\boldsymbol{\lambda}, \epsilon). \tag{6.17}$$

It has been shown (see e.g. Grothey [2001] Theorems 6.1-6.4) that as long as certain conditions hold the (global) solution to $\mathcal{F}$ is a (global) solution to $\mathcal{W}_D(\epsilon)$, and that the converse holds, namely that a (global) solution of $\mathcal{W}_D(\epsilon)$ is a (global) solution to $\mathcal{F}$. Thus we can solve the original problem by solving $\mathcal{W}_D(\epsilon)$.

With the standard augmented Lagrangian function the objective is not separable, even when $h(\boldsymbol{x})$ is separable, due to the quadratic penalty term $\epsilon h(\boldsymbol{x})^2$. When $h(\boldsymbol{x})$ is written in full (either as (6.7) or (6.8) depending on the decomposition) and squared there are cross terms involving variables from different subproblems. To be able to solve the problem by a decomposition method the subproblems must be made separable. This can be done by *diagonal quadratic approximation* as in Ruszczynski [1995], or *disaggregated simplical decomposition* as in Larsson and Yuan [1994]. A more common method, and the one we will use, is the *coordinate axes search*, also seen in e.g. Cohen and Li Zhu [1984], Gunn [1988], Kontogiorgis and Meyer [1998] or Nishi et al. [2005]. In this method we linearise the cross terms in $h(\boldsymbol{x})$ around a tentative guess at the optimal solution, for which the last solution value is used. When solving subproblem i all shared variables $\boldsymbol{t}^{i'}, i' \neq i$, are held fixed at their last values. The method is therefore similar to the coordinate axes search from pushing individual columns in the simplex method, from Section 3.2.3.

We showed in Section 3.2.3 how a coordinate axes search can stall if there are break points in the response curve. Here we are using a shared quadratic penalty that is smooth and convex, so the coordinate axes search will not stall. The global minimum is found each settling iteration if the subproblems are convex, else a local minimum is still found (Cohen and Li Zhu [1984]). In Section 6.2.5 we look at replacing the shared quadratic penalty with piecewise linear sections. These sections are dynamically updated so that they smoothly model the quadratic and the coordinate axes search does not stall. This is motivated by the fact that SLP has proved more effective than SQP for solving GPPs (see Section 5.1.1 of Chapter 5), hence we would like to keep the problems linear rather than quadratic to maintain this efficiency.

The coordinate axes search may require many solves of each subproblem to *settle*. This is necessary due to the non-separability of the augmented Lagrangian objective function. To distinguish it from overall convergence, we call solving $\mathcal{W}(\hat{\boldsymbol{\lambda}}, \epsilon)$ for a given (but perhaps not optimal) $\hat{\boldsymbol{\lambda}}$, *settling*. Before a solve starts ϵ is set to a positive constant value. However we will see later that it can be beneficial to vary ϵ during a solve. The problem is then solved by a series of *settling iterations*. Table 6.5 gives one settling iteration of the augmented Lagrangian

algorithm. Note that in Step 2 there are potentially many solves of each problem, before the subproblems have *settled*. For purposes of comparison with Benders decomposition and the dual cutting plane method we still refer to each solve of all the subproblems as an outer iteration.

1 Fix the costs $\boldsymbol{\lambda}$.
2 Solve $\mathcal{W}(\hat{\boldsymbol{\lambda}}, \epsilon)$ by minimising $\boldsymbol{x}$. This involves solving the NLP subproblems, until they have *settled*. Each solve of all subproblems once each is called an outer iteration.
3 Update $\boldsymbol{\lambda}$, by an augmented Lagrangian update formula.

Table 6.5: A settling iteration of augmented Lagrangian method

Alternative implementations to the one given in Table 6.5 exist where $\boldsymbol{x}$ and $\boldsymbol{\lambda}$ are adjusted simultaneously (see e.g. Mangasarian [1975]), and where $\boldsymbol{\lambda}$ is refined before the optimal $\boldsymbol{x}$ is found (see e.g. Tripathi and Narrendra [1972]) for that $\boldsymbol{\lambda}$.

In Step 3 $\boldsymbol{\lambda}$ is updated by any of the possible augmented Lagrangian update formulas. At the solution to $\mathcal{W}(\boldsymbol{\lambda}, \epsilon)$, $\boldsymbol{h}$ will often be nonzero, despite the quadratic penalty. The value of $\boldsymbol{h}$ in the solution indicates how $\boldsymbol{\lambda}$ must be moved to create zero violation. If $\boldsymbol{h}$ is being held at the solution by the quadratic penalty, and the dual response curve is linear with no break points between the old value of $\boldsymbol{\lambda}$ and the proposed new value, then the optimal $\boldsymbol{\lambda}$ is equal to μ_{HP}, defined in the formula below:

$$\mu_{HP}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) = \hat{\boldsymbol{\lambda}} + 2\epsilon\boldsymbol{h}. \tag{6.18}$$

This formula is well known as the standard augmented Lagrangian update formula. It can be used to update $\boldsymbol{\lambda}$ using only information from the last settling (i.e. the value of $\boldsymbol{h}$), and takes the place of the master problem that was used in the dual cutting plane method. This form is shown in e.g. Rockafellar [1976], Glad [1979], and Bertsekas [1982] to lead to the optimal $\boldsymbol{\lambda}$ with at least linear convergence for a suitable starting point and large enough ϵ. Of course, sometimes $\boldsymbol{h}$ is not being held at the last solution by the quadratic penalty, or the dual response curve is nonlinear or has break points. In this case several updates using μ_{HP} are required to find the optimal $\boldsymbol{\lambda}$. We will use a modified version of this formula to update $\boldsymbol{\lambda}$. Note that the dual cutting plane with quadratic penalty method of Section 6.2.3 may be thought of as a special case of the augmented Lagrangian method, which updates $\boldsymbol{\lambda}$ using information from all previous settlings.

The standard update is repeated along with several other possible update formulas below, the origins of which are all documented in Tapia [1977]. $\hat{\boldsymbol{\lambda}}$ is the current value of $\boldsymbol{\lambda}$, $\boldsymbol{h_x}$ is the gradient of $\boldsymbol{h}$, $\boldsymbol{f_x}$ is the gradient of $f(\boldsymbol{x})$, and $\boldsymbol{H} = (\nabla^2_{xx} w(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon))^{-1}$, the inverse of the Hessian of $w(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon)$.

$$
\begin{aligned}
\mu_{HP}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) &= \hat{\boldsymbol{\lambda}} + 2\epsilon \boldsymbol{h}, \\
\mu_{P}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) &= -[\boldsymbol{h_x}^T \boldsymbol{h_x}]^{-1} \boldsymbol{h_x}^T \boldsymbol{f_x}, \\
\mu_{B}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) &= \hat{\boldsymbol{\lambda}} + [\boldsymbol{h_x}^T \boldsymbol{H} \boldsymbol{h_x}]^{-1} \boldsymbol{h}, \\
\mu_{*}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) &= [\boldsymbol{h_x}^T \boldsymbol{H} \boldsymbol{h_x}]^{-1} [\boldsymbol{h} - \boldsymbol{h_x}^T \boldsymbol{H} (\boldsymbol{f_x} + 2\epsilon \boldsymbol{h_x} \boldsymbol{h})].
\end{aligned}
\tag{6.19}
$$

Tapia [1977] shows that all these update formulas can be written in the form below, where $\boldsymbol{A}$ and $\boldsymbol{D}$ are general matrices and $\boldsymbol{w_x}$ is the gradient of $w(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon)$.

$$
\mu_{GUF}(\boldsymbol{x}, \hat{\boldsymbol{\lambda}}, \epsilon) = \hat{\boldsymbol{\lambda}} + [\boldsymbol{h_x}^T \boldsymbol{D} \boldsymbol{h_x} + \boldsymbol{A}^{-1} [\boldsymbol{h} - \boldsymbol{h_x}^T \boldsymbol{D} \boldsymbol{w_x}].
$$

Dual response curve

The augmented Lagrangian dual response curve is relevant for two reasons. Firstly understanding why dual solutions to the augmented Lagrangian are primal feasible. Secondly it gives motivation for the proposed new $\boldsymbol{\lambda}$ update formula of Section 6.2.5.

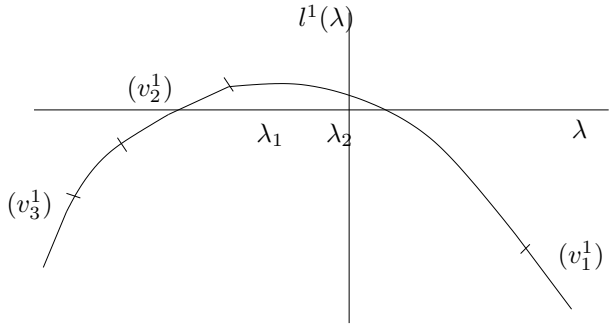


Figure 6.12: Augmented Lagrangian dual response curve of Ex2

Figure 6.12 shows the augmented Lagrangian dual response curve of example factory Ex2, for comparison with the Lagrangian dual response curve of Figure 6.9(b). The positions of λ_1 and λ_2 in each graph correspond. In the augmented Lagrangian dual response curve the linear sections, labelled v_1^1, v_2^1 and v_3^1, are now separate from each other, as due to the addition of the quadratic penalty it now requires a greater $\boldsymbol{\lambda}$ to have the same effect on the primal violation. Note that in Figure 6.12 at the optimal value of $\boldsymbol{\lambda}$ (where $v^1 = 0$), v^1 is not degenerate.

Between each linear section is a curved section. On the curved sections the quadratic penalty is active in affecting the constraint violation. A small shift in $\boldsymbol{\lambda}$ results in a small shift in $\boldsymbol{v}$, therefore also a shift in $\boldsymbol{h}$ and $\epsilon \boldsymbol{h}^2$. Thus a change in $\boldsymbol{\lambda}$ changes the quadratic penalty, which is the normal state of affairs for an augmented Lagrangian method. In this case the size of $\boldsymbol{h}$ indicates how far $\boldsymbol{\lambda}$ must be moved to make $\boldsymbol{h} = \boldsymbol{0}$, so the standard $\boldsymbol{\lambda}$ update of (6.18) works well.

On the straight sections of Figure 6.12 the quadratic penalty is not active in affecting the constraint violation. A small shift in $\boldsymbol{\lambda}$ will not immediately change $\epsilon \boldsymbol{v}$ or $\epsilon \boldsymbol{h}$ or $\epsilon \boldsymbol{h}^2$. Thus a change in $\boldsymbol{\lambda}$ does not affect the quadratic penalty, so applying the standard update formula does not work well.

Let us define the *rate of curvature* as the speed of turn of a curve, example x^2 has a rate of curvature of 2. by that definition the rate of curvature of each curved section is $\frac{1}{2\epsilon}$. So increasing ϵ has the effect of lessening the rate of curvature, hence widening the curved sections, meaning the standard update is effective more often, and fewer settling iterations are needed to find the optimal $\boldsymbol{\lambda}$. This leads to the observation below, and gives the motivation for the proposed new $\boldsymbol{\lambda}$ update formula of Section 6.2.5.

Observation 5. *Augmented Lagrangian requires few settlings with a large value of* ϵ.

Recall the total number of outer iterations is the number of settling iterations times the average number of outer iterations per settling. This observation does not say anything about the total number of outer iterations, only the number of settling iterations.

Having seen the augmented Lagrangian dual response curve we have a new interpretation for cuts generated by the dual cutting plane with quadratic penalty method. When non tight cuts, such as those shown in Figure 6.11, are shifted down by $\epsilon \boldsymbol{v}^2$, they are tangential to points on the curved sections of the augmented Lagrangian dual response curve.

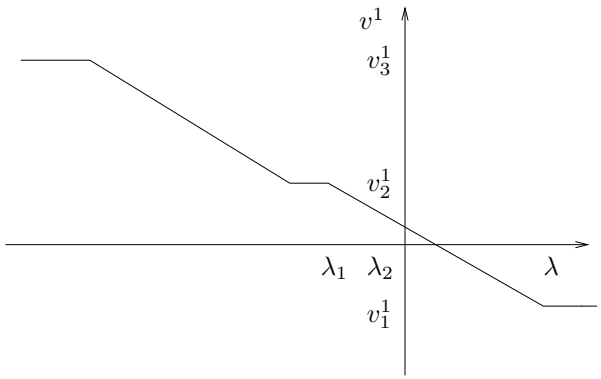


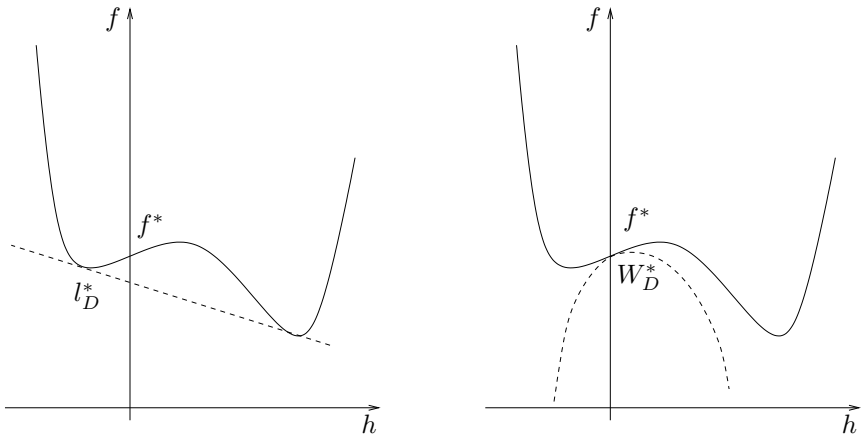
Figure 6.13: (λ, v^1) graph for augmented Lagrangian formulation of Ex2

Figure 6.13 shows the relationship between v^1 and $\boldsymbol{\lambda}$ in the augmented Lagrangian formulation of Ex2, for comparison with their relationship in the standard Lagrangian shown in Figure 6.10. In the augmented Lagrangian graph there are diagonal slopes where there were vertical sections before. Each slope has gradient proportional to $\frac{1}{\epsilon}$. On these sloped sections, including the one with the optimal $\boldsymbol{\lambda}$, $\boldsymbol{h}$ is not degenerate. The absence of primal degeneracy is an advantage of augmented Lagrangian over standard Lagrangian.

Size of quadratic penalty ϵ

The quadratic penalty is what separates augmented Lagrangian from standard Lagrangian. We consider first the effect of adding a quadratic penalty, then the importance of the size of the penalty.

There are the three benefits to having a quadratic penalty. Firstly, as we saw above when considering the dual response curves, a quadratic penalty can mean that the primal solution is not degenerate. Secondly, the linking constraint violation is an indicator of the pressure against the quadratic penalty, which indicates how to update $\boldsymbol{\lambda}$.



(a) (b)

Figure 6.14: Primal-dual response curve (solid lines) and Lagrangian approximations (dashed lines) for example problem

Thirdly, as mentioned by Hestenes [1969], Rockafellar [1974], Mangasarian [1975] and Bertsekas [1982], by choosing ϵ above some constant a problem can be made convex around the solution, removing the duality gap. We call this constant ϵ^{min}. Figure 6.14, adapted from Grothey [2001], shows the relationship between f and $\boldsymbol{h}$, which we refer to as the *primal-dual response curve*, for an example problem. In Figure 6.14(a) a standard Lagrangian penalty has been applied. At the optimal value, where $\boldsymbol{h} = \mathbf{0}$, there is a duality gap between the Lagrangian penalty and the primal-dual response curve. In Figure 6.14(b) an augmented Lagrangian penalty is applied to the same problem. If $\epsilon \geq \epsilon^{min}$, as it is here, there is no duality gap. The diagram shows pictorially that the more concave the relationship between f and $\boldsymbol{h}$ is at the solution the larger a value of ϵ is needed. In Bertsekas [1982] ϵ^{min} is shown to be the size of the largest negative eigenvalue of the Hessian of $l(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$.

We now consider the importance of the size of ϵ. Although it has no effect on theoretical convergence (as long as $\epsilon \geq \epsilon^{min}$), the size of ϵ can have a large effect on speed of convergence. Many augmented Lagrangian algorithms heuristically increase ϵ every settling iteration, see e.g. Miele et al. [1971], Tapia [1977], Gunn [1988], Kontogiorgis and Meyer [1998] and Larsson

and Yuan [1994]. In Glad [1979] and Conn et al. [2000] after each settling iteration either $\boldsymbol{\lambda}$ is updated, or ϵ is increased.

By Observation 5, large values of ϵ can lead to fewer settling iterations, as each time $\boldsymbol{\lambda}$ is moved it is moved more substantially. However, in Hestenes [1969] a large value of ϵ is shown to be undesirable when working by hand as it causes rounding errors. In modern solves a large ϵ is also undesirable as it can cause ill conditioning leading to numerical errors. Furthermore, when using a coordinate axes method, a large quadratic penalty constrains each subproblem to be close to its previous solution, causing small steps and slow settling (recall settling is solving $\mathcal{W}(\boldsymbol{\lambda}, \epsilon)$ for a given $\boldsymbol{\lambda}$). This leads to the observation below, which can be seen as the counterpart to Observation 5.

Observation 6. *Each settling iteration takes many outer iterations with a large value of ϵ.*

Observation 5 suggests a large value of ϵ should be used, as it means few settlings are required. Observation 6 suggests a small value of ϵ should be used, as each settling iteration then requires few outer iterations. This conflict is the motivation for the new augmented Lagrangian update formula of Section 6.2.5.

6.2.5 Augmented Lagrangian algorithm

This section describes the algorithm, called AAL1, that we use to solve the multifactory problem. We use the standard augmented Lagrangian function and a modified form of the standard update for $\boldsymbol{\lambda}$. Although augmented Lagrangian is not a new solution approach we use some novel modifications here.

For most multifactory problems we use the row linked form of Split 1, and linking constraints of the form of (6.7). For multifactory problems with simple transport problems we use the row linked form of Split 2, and linking constraints of the form of (6.8). In this case each linking constraint features variables from all the factories, so each settling iteration may take many outer iterations.

We solve the transport problem then we solve all the factories, then repeat this cycle of $1 + F$ problems until settling, at which point $\boldsymbol{\lambda}$ is updated. We solve each subproblem by the SLP solver Slp5, with a fairly loose tolerance on convergence. $\boldsymbol{\lambda}$ is initialised by setting it to the smallest value that could ever be optimal. $\boldsymbol{t}$ is initialised by sharing out the maximum use of each shared raw equally among the factories. ϵ is typically initialised at a value of 1, and increased by a factor of 1.2 each outer iteration.

There are three innovations in the algorithm, compared to a standard augmented Lagrangian algorithm. The first two concern updating $\boldsymbol{\lambda}$, the third one is to replace the quadratic penalty in the subproblems with piecewise linear sections. These innovations are explained in the two subsections below. We then analyse the efficiency of the algorithm and present results.

Updating $\boldsymbol{\lambda}$

The first innovation that we use in algorithm AAL1 is a new update formula for $\boldsymbol{\lambda}$. This is a change to Step 3 of the algorithm given in Table 6.5. This aims to get the benefit of few settling iterations that occurs with a large ϵ, without any of disadvantages.

$$\begin{aligned} &\min_{\boldsymbol{t}} -100t^1 + 80t^2 \\ &\text{s.t. } t^1 - t^2 = 0 \\ &\text{s.t. } 0 \leq t^1, t^2 \leq 1. \end{aligned} \tag{6.20}$$

To motivate the new update we give an example where the standard update does poorly. Ex3, shown above, is a simple linear problem with two variables, each with objective costs and simple bounds. The only constraint is one that links the two variables, $\boldsymbol{h} = t^1 - t^2 = \mathbf{0}$. In keeping with the notation of the multifactory problem we distinguish the variables with superscripts, rather than subscripts, so we can think of t^1 and t^2 as being two variables from different subproblems. We form the augmented Lagrangian model of Ex3 by relaxing the constraint and adding both a linear and quadratic penalty on its violation to the objective. The resulting problem is:

$$\begin{aligned} &\min_{\boldsymbol{t}} -100t^1 + 80t^2 + \lambda(t^1 - t^2) + \epsilon(t^1 - t^2)^2 \\ &\text{s.t. } 0 \leq t^1, t^2 \leq 1. \end{aligned} \tag{6.21}$$

The optimal solution to (6.21) is $t^1 = t^2 = 1$, with $\boldsymbol{\lambda}$ anywhere between 80 and 100 giving this solution. We consider solving Ex3 using augmented Lagrangian, starting with $\boldsymbol{\lambda} = \mathbf{0}$.

Firstly, suppose we use the standard update and a fixed value of $\epsilon = 1$. In the solution to each subproblem $t^1 = 1, t^2 = 0$, so the simple bounds, not the quadratic, are active in constraining $\boldsymbol{h}$. Thinking of Figure 6.12, this is finishing on a straight section of the dual response curve, where we have seen that the standard update does not do well. On successive updates $\boldsymbol{\lambda}$ goes $0, 2, 4, 6, \ldots, 76, 78, 80$, taking 40 iterations to converge.

Secondly suppose we instead use the standard update and a fixed value of $\epsilon = 50$. The optimal $\boldsymbol{\lambda}$ is found in one iteration. At the solution to the first t^2 subproblem $t^2 = 0.2$, away from its simple bounds as it is being held by the large quadratic penalty. Thus $\boldsymbol{h}$ has been constrained by the quadratic penalty. Using a large value of ϵ has widened the curved sections of the dual response graph, so the start point $\boldsymbol{\lambda} = \mathbf{0}$ now falls on a curved section, for which case the standard update works well.

Thirdly we consider using a new update formula, μ_{HP2}, that uses $\boldsymbol{h}_D$, the dual of the linking

constraint. For the decomposition of Ex3 $\boldsymbol{h}_D$ is given by the smaller of the dual costs on t^1 and t^2.

$$\mu_{HP2}(\boldsymbol{x}, \boldsymbol{\lambda}, \epsilon) = \boldsymbol{\lambda} + 2\epsilon \boldsymbol{h} + \boldsymbol{h_D}. \tag{6.22}$$

$\boldsymbol{h}_D$ is zero when at least one variable is being held by the smooth quadratic penalty, in which case the standard update formula will already work well. However, when neither variable is being held by the quadratic penalty, $\boldsymbol{h}_D$ is a substitute for measuring what the pressure would be on the quadratic penalty if it was active. In this case using the dual cost replicates the effect of large ϵ, allowing μ_{HP2} to move $\boldsymbol{\lambda}$ further and improve the speed of the solve. If we use μ_{HP2} then even with a fixed value of $\epsilon = 1$ the optimal $\boldsymbol{\lambda}$ for Ex3 is found in one iteration. Solving the multifactory problem G takes 19 outer iterations using AAL1 with μ_{HP}, and 16 outer iterations using AAL1 with μ_{HP2}.

It is a feature of some decomposition algorithms to solve some early subproblems only approximately, as long as some measure of progress is made (see e.g. Ruszczynski [1995], Larsson and Patriksson [1995]). This is a reasonable approach as in early settling iterations when $\boldsymbol{\lambda}$ is not close to optimal the exact value of the subproblem solves is not so important. Thus the second innovation in AAL1 is to limit each settling iteration to just one outer iteration, regardless of whether or not $\boldsymbol{x}$ has settled. This is a change in Step 2 of the augmented Lagrangian algorithm of Table 6.5. Although there is a potential risk that in doing so we may not make a good update of $\boldsymbol{\lambda}$, in practice this innovation worked well. For example, solving the multifactory problem G takes 39 outer iterations using AAL1 without limiting the settling iterations, and 16 outer iterations when settling iterations are limited to one outer iteration each.

Modelling quadratic penalty

Each subproblem is an NLP, which we solve by repeatedly linearising the NLP in an SLP algorithm. With each linearisation the quadratic penalty of the augmented Lagrangian becomes just a single tangent line, which only has the correct gradient at the point of linearisation. The third innovation in AAL1 is to replace the quadratic penalty in each subproblem with a piecewise linearisation (PL), which can model the quadratic penalty more accurately than a single tangent line. We use a PL with five sections, spanning the trust region from $x_i - \rho$ to $x_i + \rho$ for each variable x_i. The sections are of equal size, and the central of the five sections has the same gradient as the quadratic at the point where the linearisation is formed. Using a PL with more sections will more accurately model the quadratic curve, and decrease the number of SLP iterations, but will also increase the number of LP iterations.

An interesting alternative spacing of the sections, that is not yet used in AAL1, is to place

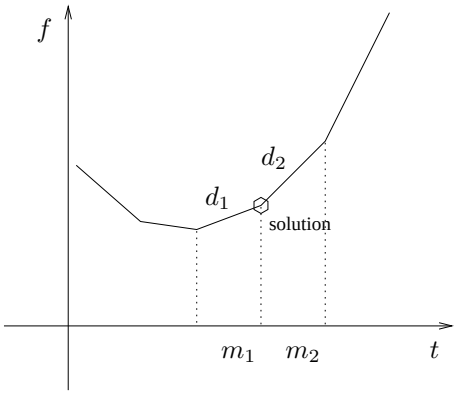


Figure 6.15: Solution at break point of PL sections

a break point at an estimate of t^*, the optimal value of t. This means that in the next LP solve t can go directly to what is believed to be the optimal value. Consider the PL shown in Figure 6.15. If t finishes at the edge of the linearisation, then the best estimate for t^* is simply $\hat{t}$, the last value of t. However, if t finishes at a break point between two sections, as in the diagram, t^* can be estimated from the mid points of the sections adjacent to the solution, m_1 and m_2, and the size of the duals on those sections, d_1 and d_2.

$$t^* \simeq \frac{-d_1 m_2 + d_2 m_1}{-d_1 + d_2}. \tag{6.23}$$

Using a spacing of sections that has a break point at the estimate for t^* allows rapid convergence of the SLP when the estimates are correct, as was often found to be the case in limited testing. This spacing of sections can also be useful in algorithms that terminate the SLP after only a few iterations, as it can be used to improve the values of the shared raws found at the end of each SLP solve.

Results

We now present results for solving multifactory problems by the augmented Lagrangian algorithm AAL1, described above. For the large P_e problem we also look in more detail at the progress of a solve, and compare the total work required with the total work required by a direct solve by SLP with no decomposition.

Table 6.6 gives results for the small test problems (before the line), then the large test problems (after the line). For each problem we restate the problem dimensions, including the size of $\boldsymbol{\lambda}$, which is equivalent to the number of shared raws. We give the number of outer iterations of AAL1, the final objective value and the error in this objective value. These errors are due to the large quadratic penalty that is present at the end of the solve, which force the linking constraints to become satisfied and the problem declared optimal without having found

	F	R	$\lvert\boldsymbol{\lambda}\rvert$	outer iterations	objective value	error
G	3	2	6	16	45.179	0
G_l	3	2	6	19	43.067	0
G_e	3	2	6	11	45.033	-0.002
G_{le}	3	2	6	18	43.067	0
G_s	3	2	2	18	42.408	0
H_e	3	4	12	31	45.993	0.005
J_e	6	2	12	30	65.251	0.002
P_e	12	4	48	35	239.18	0.02
Q_e	12	2	24	13	228.55	0.02
R_e	3	7	21	25	64.794	0.02

Table 6.6: Results for AAL1

the absolutely correct $\boldsymbol{\lambda}$. Although it is possible to modify AAL1 so it continues until the errors are reduced, this takes many more outer iterations. Note that only the outer iterations are given, because of the quadratic penalty making the problems more nonlinear AAL1 tends to take slightly more SLP and LP iterations per outer iteration than the other methods.

The problem versions that are the same size as each other, but have different ways of supplying the shared raws (G, G_l, G_e and G_{le}), all solve in a similar number of outer iterations. Among all problems the ones with more $\boldsymbol{\lambda}$ variables tend to take more outer iterations, though the number of outer iterations is not much longer for the larger problems, which suggests that AAL1 scales well with problem size.

We now summarise results for the three decomposition methods that we have used on the multifactory problem. Benders decomposition was unable to solve large the problems. The dual cutting plane method solved all problems, and was very fast for problems with few linking variables, namely G_s and P_s. But for large problems with many shared raws augmented Lagrangian was by far the best method.

Figure 6.16 shows more details of the AAL1 solve of P_e. Figure 6.16(a) shows the dual objective $l^*(\boldsymbol{\lambda})$ initially increasing as $\boldsymbol{\lambda}$ is optimised, then decreasing fractionally as the problem becomes primal feasible. Figure 6.16(b) shows the infeasibility, defined as the maximum distance between transport supply and factory use for any shared raw. The tolerance of infeasibility of linking constraints before we declare optimality is 10^{-3} (with infeasibility of -0.710^{-3} at the solution). Figure 6.16(c) shows the value of $\boldsymbol{\lambda}$ for the first factory. The $\boldsymbol{\lambda}$ variable for the second shared raw (labelled as $f1t2$) is the largest and takes the longest to converge, which is typical across all twelve factories, In fact solving P_e without this shared raw takes only eight outer iterations, compared to 35. Figure 6.16(d) shows the supply and demand of Raw 1 (top two lines) and Raw 2 (bottom two lines) from transport problem to Factory 1. Raw 2, which corresponds to the large $\boldsymbol{\lambda}$ variable in Figure 6.16(c), takes a long time to get feasible. As it becomes feasible the values for the other shared raw also move.

We now wish to compare the time taken to solve the multifactory problem by augmented Lagrangian with the time taken to solve by a direct solve. We focus on the largest and most

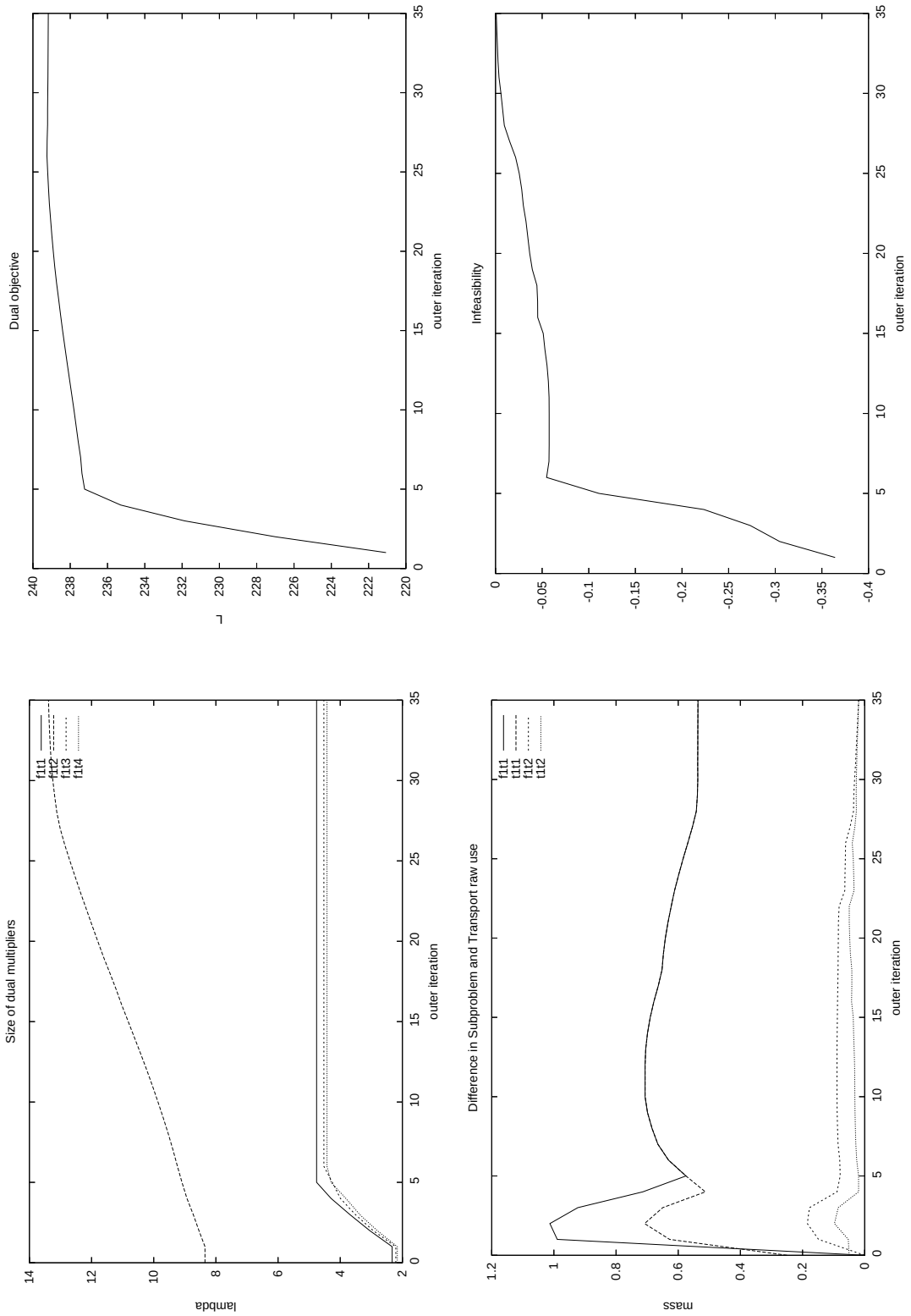


Figure 6.16: AAL1 solve of P_e, showing objective $l^*(\boldsymbol{\lambda})$ (a), infeasibility $\boldsymbol{h}$ (b), $\boldsymbol{\lambda}$ for factory one (c), $\boldsymbol{t}$ for factory one (d)

significant test problem, P_e. In order that we can properly compare the augmented Lagrangian and direct solve we must estimate the time in each solve that is spent in overheads, such as reading in and out of files. Although some overheads are unavoidable almost all of the time of a solve should be spent solving LPs, so as a guide to the efficiency we calculate the amount of time of each solve in EMSOL. By isolating the time for EMSOL for the whole solve time we arrived at the figures of 8% approximate efficiency for the augmented Lagrangian solve and 70% approximate efficiency for the direct solve. The reason the augmented Lagrangian solve is so inefficient is because the SLP solver spends a significant amount of time in each SLP solve calculating the gradients of the nonlinear constraints, and there are many (small) SLPs to be solved in the decomposition. A specialised solver like Integra_1, which recognises the constraints of the pooling problem and automatically calculates the gradients, would be much more efficient. Although Slp5 is also used for the direct solve as there is only one (large) SLP to solve here the proportion of time getting gradients is much smaller.

	outer iterations	SLP iterations	LP iterations	(adjusted) time
Direct solve	-	10	70,000	390
AAL1	35	1,300	57,000	87

All the small multifactory problems solve quickly with a direct or decomposed method, so no comparison of times is given. In the table above we compare the solve of P_e with a direct solve and with AAL1. AAL1 takes many more SLP iterations than the direct solve, though almost all of these are warm started, so require very few LP iterations. In total AAL1 takes about as many total LP iterations as the direct solve. This indicates that in fact AAL1 is doing considerably less total work, as each LP iteration in AAL1 is on a far smaller LP than an LP iteration in the direct solve, as it is on a subproblem and not the whole multifactory problem. The times, adjusted to reflect the efficiency estimates for each code, reflect this, with AAL1 much faster than the direct solve.

6.3 Finding global minimum

All of the previous methods we have seen to solve multifactory problems; Benders decomposition, dual cutting plane method, augmented Lagrangian and direct solve by SLP, work best when the problems are convex. When the problems are nonconvex there is a danger of the solution finding only a local minimum, or even failing to converge. These local minima could have considerably worse objective than the global minimum (though we found all the local minima were close in objective value in the GPP test problems of Chapter 5 shown in Figure 5.6 and Figure 5.7).

In Chapter 5 we discussed how many solves from random start points are needed to give a good chance of finding the global minimum of a GPP. In this section we give two new algorithms that actively attempt to find the global minimum of multifactory problems.

The GPP and multifactory problems are both NLPs. There are various methods to find the global minimum of an NLP, outlined in Archetti and Schoen [1984]. In *space covering techniques* (also called spatial branch and bound) the solution space is subdivided and local solves are done on many of the the subdivisions, which eventually guarantees to find to global minimum. In a *scatter search* method the chance of finding the global minimum is increased by generating many random start points and performing local optimisation from the best of these. Rotondi and Drappo [1995] describe a scatter search method as having four stages. In the *sampling* stage initial sample points are generated. In the *classification* stage sample points are classified as either promising or not. In the *local search* stage a local optimisation is performed from promising sample points. The *stopping* stage determines when to stop the algorithm.

As an aside, we can view the Integra_1 algorithm as a sort of degenerate scatter search, with the four stages above. Sampling consists of generating random start points for each of the many solves in a run. Classification is trivial in Integra_1; all random start points are selected to be locally optimised from. Local search is by SLP, and stopping is after a fixed number of solves are completed, though alternative stopping rules were discussed in Section 5.4.1.

We now introduce two new algorithms to find the global minimum of a multifactory problem. These are both scatter search methods, that categorise sample points using the idea of *clustering* (Torn [1976]). The purpose of clustering is to recognise which start points in the solution space lead to the same local minimum, to avoid having to do random starts from all of them. Similar methods to avoid redundant solves are seen in Rinnooy Kan and Boender [1987] and Locatelli and Schoen [1999].

Cluster

A *cluster*, in the context of the multifactory problem, is a section of the solution space for a factory. We can think of each cluster as being a factory local minimum and all the point close to that local minimum. A *representation* is a particular instance of a cluster, so is a specific value for the variables of the factory. A *cluster combination* is a choice of one of the possible clusters for each factory. The clusters are so defined so that a multifactory solve from a given cluster combination will lead to a given local minimum, independent of the choice of representations for each cluster.

We make two assumptions; firstly that we can group the different factory solutions into clusters based on solves of the multifactory problem, and secondly that a solve warm started from a cluster combination will finish with each factory in the same cluster it started in. If these assumptions do not hold our methods may still have value, and by using them we may still see an improvement on simply doing repeated random start solves of the multifactory problem. We think of the clusters for each factory as *equivalence classes* that partition the solution space, so every solution point is a member of one and only one cluster. Figure 6.17 gives a (hypothetical)

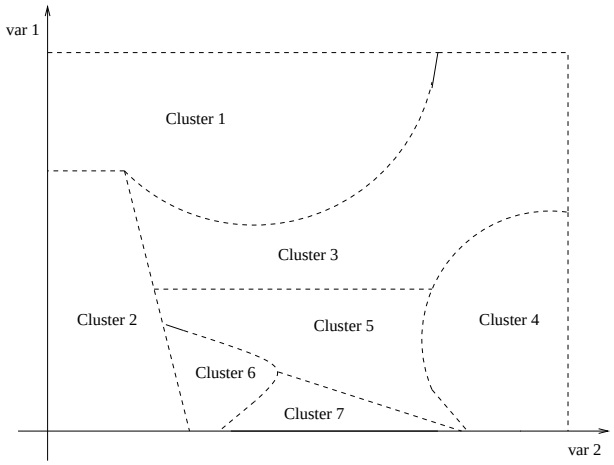


Figure 6.17: Example 2D solution space broken divided into clusters

division of the solution space into clusters for a problem with only two variables. Depending on where the start point is in the space a different local minimum will be reached.

Both the algorithms we introduce algorithms are novel in relying on the near separability of the multifactory problem into factories, so that it can be decomposed. In Section 6.3.1 we describe an algorithm intended for problems with few shared raws. In Section 6.3.2 we describe a less exact algorithm intended for problems with many shared raws, which are therefore more difficult.

6.3.1 ILP model algorithm

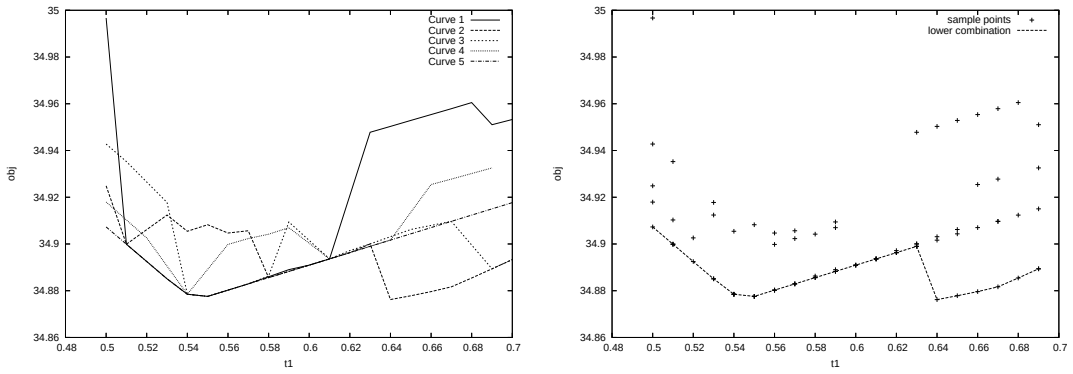
The ILP model algorithm attempts to find the global minimum of multifactory test problems. Although effective on small test problems this algorithm is not very practical, and is intended mostly as an introduction to the more useful algorithm of the next section.

We attempt to identify all the clusters of each factory, and find the associated shared raw response curve for each of them. We call these the *local response curves* (LRCs) of a factory. These can then be combined (as we will see below) to form what we call the *global response curve* (GRC) of a factory, which completely describes the factory. When the GRCs of all the factories are combined with the entire transport problem it gives a model for the entire multifactory problem. This model can then be solved, and may give the global minimum to the multifactory problem, or at least suggest the correct cluster of each factory, which can then be used as a warm start point for a new NLP solve. Table 6.7 gives an outline of the algorithm.

1	Find LRCs for each factory.
2	Combine these to find GRC for each factory.
3	Add GRCs of each factory to transport problem, and solve this ILP to find a cluster combination.
4	Warm start the original NLP from this cluster combination.

Table 6.7: ILP model algorithm

In Step 1 finding all the clusters and their associated LRCs can be time consuming, so is worthwhile only when there are relatively few shared raws. For some factories the LRCs may be known in advance, if for example similar problems have been solved in the past. If they are not known one method to find them is to solve the factory many times, each time fixing the values of the shared raws at different sample values, and record the objective value and basis at each points. Joining up points with similar bases gives a linearised approximation of the LRCs, which is used in practice. If the LRCs are convex they can be modelled linearly, if they are concave then integer variables are used.



(a) (b)

Figure 6.18: Five LRCs (a) and GRC (b) for Sonoco97 from using algorithm ALC1

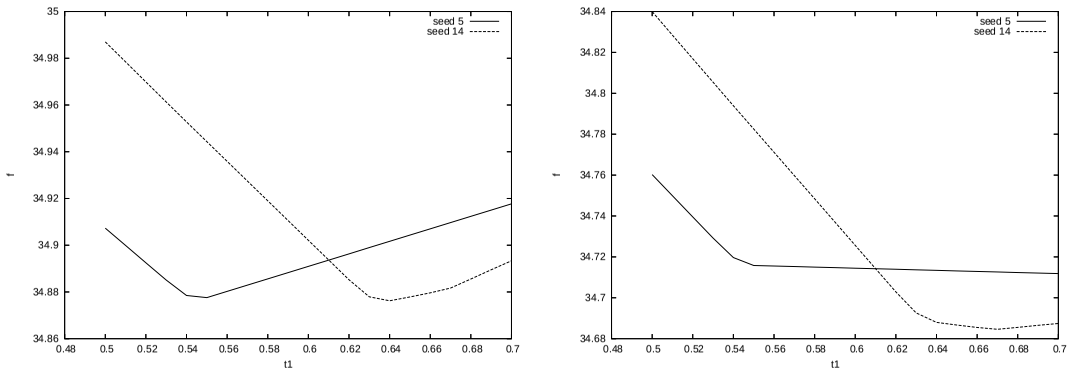
In Step 2 we combine the LRCs. We developed an algorithm, ALC1, to find the GRC for a factory out of the LRCs, for problems with one shared raw. Figure 6.18(a) shows five LRCs for Sonoco97 (for one shared raw). Each LRC has a different local minimum. There are points on each of the LRCs that are never optimal, as the other response curve uses the same amount of raw with lower objective value. Figure 6.18(b) shows the GRC, formed from taking sections from each of the LRCs. The GRC model will contain any integer variables that were required to model concave LRCs, as well as possible extra integer variables for new concavity that has been introduced. The GRC model is an ILP approximation of the actual NLP response curve.

If it is accurate a GRC like Figure 6.18(b) is an excellent model for a factory. The factory is reduced to being of dimension equal to the number of shared raws (one in the case of Figure 6.18(b)). The GRC is similar to the master problem of Benders decomposition, but here we have the distinct advantage that our small model of each factory can be nonlinear.

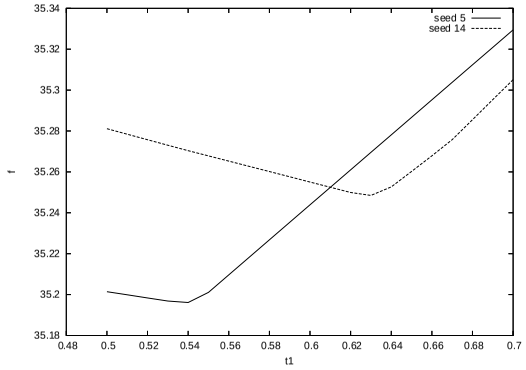
In Step 3 we combine the ILP model for each factory with the entire transport problem to give an ILP that approximately models the whole multifactory problem. The value of the integer variables at the solution to this ILP gives a *cluster combination* for the multifactory problem. In Step 4 we warm start the original NLP from this combination. This can lead to the global minimum, if the clusters are well modelled and so the approximate ILP solution is

close to the NLP solution.

Results



(a) (b)



(c)

Figure 6.19: LRCs for factories of Sonoco97_1

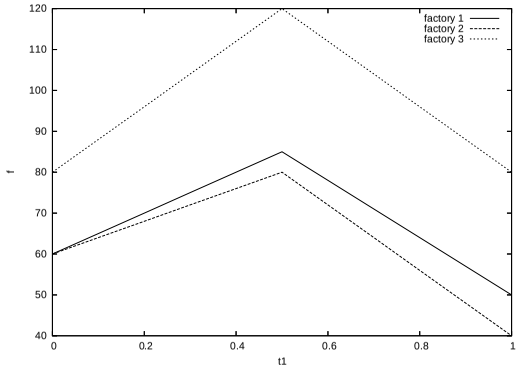


Figure 6.20: LRCs for factories of v1v_1

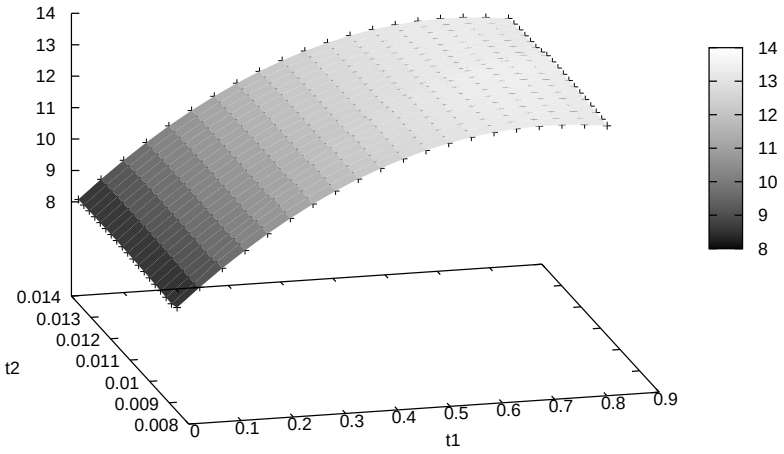


Figure 6.21: LRC for one factory of G1bConc_2

Test problems

For three multifactory problems we modelled the LRCs of each factory, then modelled and solved the whole multifactory problem as an ILP. In each case the LRCs were found by a heuristic examination of each problem (not our algorithm ALC1). Figure 6.19 shows two LRCs for each of the three factories of Sonoco97_1. For each factory we show only the two LRCs that are not dominated by any others. The GRC is therefore made out of these two LRCs for each factory. Figure 6.20 shows the concave response curves to all three factories of v1v_1. Each response curve can be divided into two linear sections, which each correspond to one LRC. Figure 6.21 shows the two dimensional response curve to one of the factories of G1bConc_2. This multifactory problem has two shared raws. Although not apparent in Figure 6.21, the response curve is concave in both directions. As there are two shared raws for this problem we model the LRCs with linear panels, rather than linear sections.

Results

For all three of these problems solving the ILP gave the correct cluster combination, and when the original NLP was warm started from this combination the global minimum was found. The NLP solve warm started from the chosen cluster combination was very quick compared to an NLP solve from a random start point. Rather than just warm starting from the chosen cluster combination, the ILP can also be forced to generate other cluster combinations, which can also be warm started from to see if they give a better solution to the NLP.

6.3.2 Recombination algorithm

The recombination algorithm is a second new algorithm to find the global minimum of multifactory problems. In the recombination algorithm we do not attempt to identify all the clusters for each factory. Instead we find just a few of the clusters by performing some number of random start NLP solves of the multifactory problem, and extracting the factory solutions each time. Further solves of the multifactory problem are then done, warm starting from distinct cluster combinations. Table 6.8 gives an outline of the algorithm.

1	Solve multifactory problem several times from random start points. This is the sampling stage.
2	Extract factory clusters from multifactory solves and give them scores. This is classification.
3	Warm start multifactory problems from promising cluster combinations. This is the local search stage.

Table 6.8: Recombination algorithm

This method should have two advantages over simply doing repeated random start solves. Firstly warm starting from distinct cluster combinations saves on redundant multifactory solves which lead to the same local minimum. Secondly with a good merit function for choosing which cluster combinations to start from good local minima can be found. The potential disadvantage of this method is that by combining a limited selection of clusters we may only cover a small portion of the total solution space.

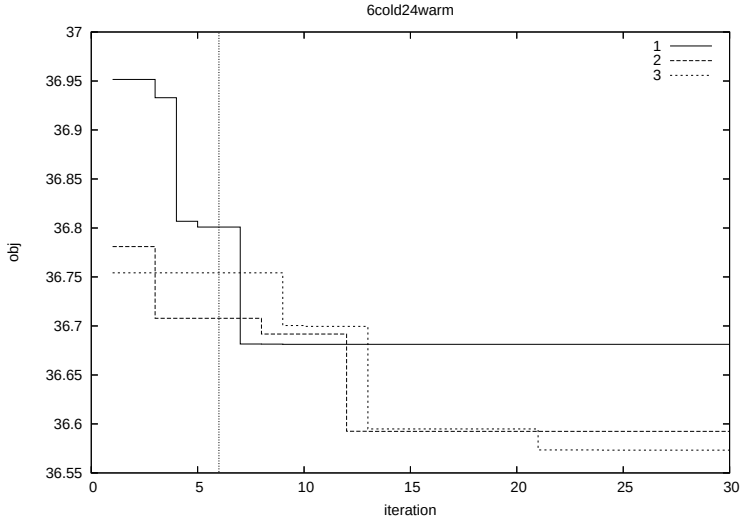


Figure 6.22: Best objective value found so far on example runs of Sept98_5

Figure 6.22 shows the decrease in best objective value for three example runs of a test problem. In each case there are six random start solves followed by 24 warm start solves. Two of the three runs finish with a point close to the known global minimum of 36.5691, and one does not.

In the recombination algorithm Step 1, the sampling stage, consists of several random start

solves of the multifactory problem, to find some of the different clusters for each factory. Any solver can be used, here for convenience we use a direct solve, applying SLP to the undecomposed problem. For larger test problems a decomposed solve might be necessary.

Step 2, the classification stage, extracts the individual factory solutions from each multifactory solve and assigns them to a cluster based on their similarity to previous solutions to that factory. We must therefore test each factory solution to see if it belongs to the same cluster as any previous factory solutions. To classify the factory solutions we first get a minimal characterisation of them. There are many equality rows in each factory, which define redundant variables. In fact the flow proportion variables, which measure the proportion of flow each bin gets from its inputs, completely describe a factory solution (see Section 5.2.1 where we eliminate all but the flow proportion variables). When comparing two different solutions to the same factory it is therefore only necessary to look at the values of the flow proportion variables. Some factories have symmetry in the bin network, for example in Sonoco97 the six mixing bins can all substitute for each other. Therefore before comparing factory solutions we permute the variables so that the mixes are in a standard order, ordering them from heaviest to lightest. This makes it possible to detect a similar factory solution that is merely a permutation of a previous solution. The alternative to ordering the variables to a solution is to add *symmetry breaking constraints* to the multifactory model before solving, to ensure the mixes finish in the standard order. This is less desirable than ordering the solutions, as the symmetry breaking constraints can lead to problems that are harder to solve. Symmetry breaking constraints are used in the silo problem of Chapter 7.

The permuted flow proportion variables give a minimal characterisation of the factory solution. There are then three tests to see if a factory solution is of the same cluster as a previous factory solution. These tests compare each factory solution with all the previous solutions to that factory. If any of the three tests is passed we declare the factory solution to belong to a previously seen cluster. If the factory solution does not match any previous solutions it becomes the first member of a new cluster.

The first classification test is comparing the values of the flow proportion variables. Any time flow proportion variable x_j^i from factory solution i and $x_j^{i'}$ from factory solution i' are not both zero or not both nonzero that is counted as one difference. If there are few enough differences between them the solutions are deemed to be from the same cluster. This test gave decisive results on the problems tested; typically when factory solutions were from different clusters there were 50 to 100 differences, when they were the from the same cluster there were only 0 to 10 differences. The second classification test is measuring the size of the sum of differences between factory solutions i and i', $\sum_j |(x_j^i - x_j^{i'})|$. If the sum is small enough factory solutions i and i' are deemed to be from the same cluster. The results of this test are usually closely correlated to the results of the first test. The third classification test is comparing the contribution to the objective value, and the mass of each of the shared raws, of each factory

solution. If there is a small enough difference between the objective value to factory solutions i and i', and a small enough difference between the mass of shared raws of factory solutions i and i', then the solutions are deemed to be from the same cluster. For multifactory problems with only one shared raw this test was able to group many factory solutions into the same cluster.

For each of the classification tests there are user defined parameters for the thresholds of when to declare two factory solutions to be from the same cluster. If the parameters make it very easy to put factory solutions into the same cluster, there is a risk of *overclassifying*, which leads to a few clusters each of which is large. Similarly the parameters can lead to *underclassifying*, which leads to many clusters each of which is small. Consider the solution space of Figure 6.17. An example of overclassifying is if points from Clusters 5 and 6 and 7 were mistakenly believed to all be from the same cluster. An example of underclassifying is if several points all from Cluster 1 were thought to belong to different classes. Rotondi and Drappo [1995] mention that creating many small clusters tends to give a greater chance of finding the global minimum, but requires more local searches.

In clustering algorithms there is a tendency for sample points (in our case factory solutions) near the edge of the feasible region to be underclassified. This is called the *edge effect* (see e.g. Ripley [1988], Rotondi and Drappo [1995]). The edge effect is not expected to be a major issue for the recombination algorithm, and we do not account for it.

Step 3, the local search stage, comes after classification. We consider which combinations of clusters to locally search from. In the scatter search algorithm of Ugray et al. [2007] they choose points to locally search from based on a *distance filter* and *merit filter*; points must be reasonably far from other points and have reasonably promising objective values. We can view our algorithm as having a distance and merit filter too. The distance filter is simply that we always locally search from a cluster combination that has not yet been used as a warm start point, and has not yet been found as a solution point (these are different tests as sometimes the NLP solves have different cluster combinations at the solution to their start points). The merit filter is that we only locally search from combinations with at least reasonably good *scores*.

The score for a cluster combination is made from the sum of the scores of each cluster. Each cluster has a score based on the factory objective value for all the factory solutions that were in that cluster, and the multifactory objective value for all the multifactory solutions of which it was a part. There is a bonus to the cluster score if the cluster is part of the current best multifactory solution found, and a penalty if the cluster is part of any infeasible multifactory solutions. High scoring clusters from each factory are then combined to give the next cluster combination. However, we do not always choose the clusters with the very highest score. The most effective method found was to have some randomness in the cluster choice, so sometimes clusters which do not have the highest scores are chosen. For a given factory there are potentially many representations known to belong to the chosen cluster, any of which can be used as a warm start point. In practice we take the specific warm start values from the first

representation that was detected from the chosen cluster.

When choosing which clusters to recombine we do not consider the gradients of shared raw use in the factory solutions, though this information is available. These gradients could be used to determine which factory solutions will combine well in the multifactory solve, but due to the nonlinearity of the factories the gradients are often misleading.

As we mentioned above the warm started local searches may have factory solutions of different clusters to their warm start points. This can make the selection of good clusters less effective, as the desired cluster combination may not be seen in the solution. However, it also means that new clusters can be found throughout the run.

Test problems

The algorithm was used to solve the multifactory problems Sonoco97_1 (that was solved before in Section 6.3.1 by the ILP model algorithm), Sonoco97_5 and Sept98_5. These each consist of three highly nonlinear factories joined by a transport problem and have one, five and five shared raws respectively. In each multifactory problem there is a limit on the mass of shared raws in the transport problem that is less than the total masses desired by each factory, and there is no way to buy extra of the shared raws. In terms of the four cost functions given in Figure 6.3 these costs are similar to those shown in Figure 6.3(b).

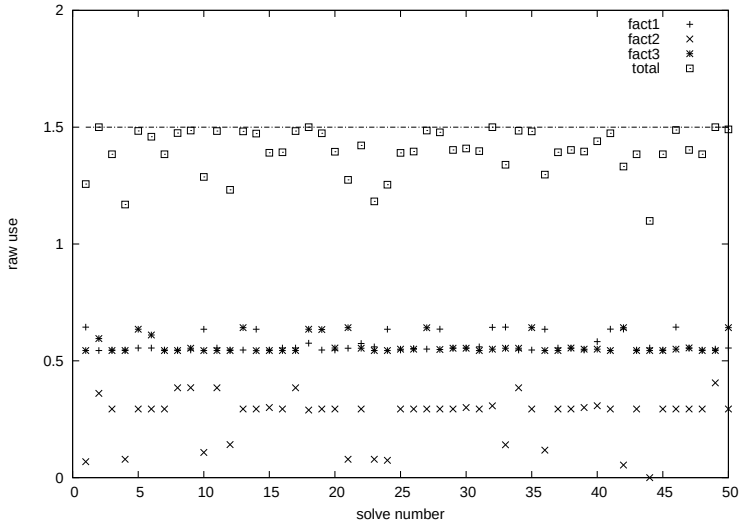
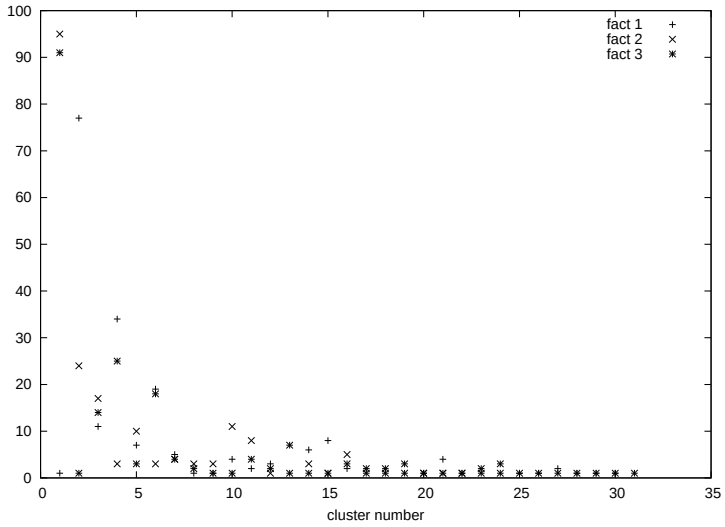


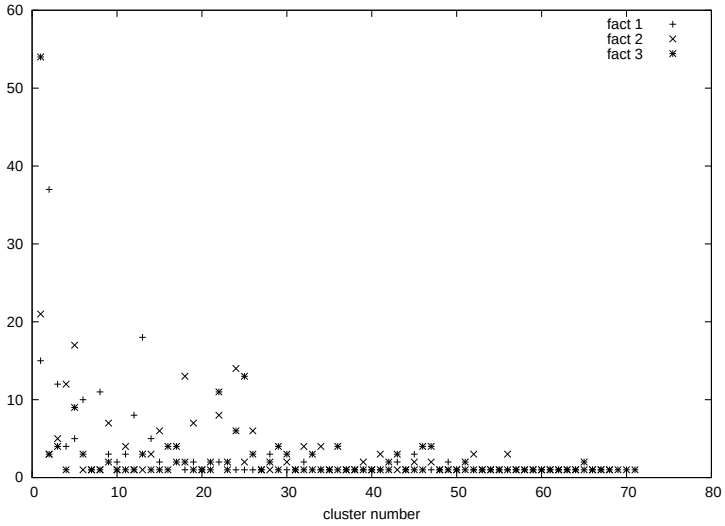
Figure 6.23: Mass of the shared raw in 50 random start solves of Sonoco97_1

We saw in Figure 6.19 that each of the Sonoco97 subfactories uses more than 0.5 units of raw material one at all the local minima. However when three Sonoco97 factories are combined and given a maximum collective allowance of 1.5 units of raw material one, they frequently do not use the full allocation. This is the case even with very cheap costs on flow in the transport problem. This indicates the extent of the nonlinearity of Sonoco97, and the prevalence of local minima. Figure 6.23 shows the mass of the shared raw used in each factory and in total in 50

random start solves of Sonoco97_1, by a direct SLP solver.



(a)



(b)

Figure 6.24: Frequency of clusters found in 200 random start solves of Sonoco97_1 (a) and Sonoco97_5 (b)

We now compare Sonoco97_1 and Sonoco97_5 to see the difference between problems with different numbers of shared raws. We first look at the random start solves. For Sonoco97_1 in 200 random start solves 137 distinct cluster combinations were found, with about 30 different clusters for each factory. Figure 6.24(a) shows the frequency of these clusters for each factory. The (suspected) global minimum was found twice in these 200 solves. An approximate probability of finding the global minimum is also given by multiplying the probabilities of each cluster finishing at the cluster of the global minimum (approximate as it assumes the factories

are independent). For Sonoco97_1 this gives the probability of finding the global minimum as $\frac{19}{200}\frac{95}{200}\frac{91}{200} = 2.05\%$

For Sonoco97_5 in 200 random start solves 195 different cluster combinations were found with about 70 different clusters for each factory. Figure 6.24(b) shows the frequency of these clusters for each factory. The global minimum was found once. Multiplying together the probabilities of the clusters of the global minimum gives $\frac{17}{200}\frac{17}{200}\frac{54}{200} = 0.20\%$. In conclusion the problem with more raws has more clusters, hence the global minimum is harder to find.

Results

	Sonoco97_1	Sonoco97_5
overclassifying	7	4
underclassifying	5	16

The table above shows the number of classification errors when classifying the factory clusters in 200 random start solves of Sonoco97_1 and Sonoco97_5. Overclassifying is detected when two multifactory solutions were thought to be from the same cluster combination, but were not, as two representations were found to have different objective value or raw use. Underclassifying is detected when two different clusters for a factory were discovered to actually be from the same cluster. For example suppose for a given factory Solution A belongs to Cluster 1 and Solution B belongs to Cluster 2, but that solutions A and B are determined by the classification tests to be different enough not to be in the same cluster. A new solve of the multifactory problem might produce Solution C, which is close to both solutions A and B (perhaps it is the midpoint of the two). This is recorded as an underclassification error and Cluster 1 and Cluster 2 are merged.

The table shows there are fairly few classification errors of either type. With more shared raws underclassifying is a greater problem. This is expected as with many shared raws it is harder to identify that two solutions belong to the same cluster.

	random start solve			warm start solve		
	SLP its	LP its	time	SLP its	LP its	time
Sonoco97_1	53.8	41,249	24.7	33.8	9,325	5.8
Sonoco97_5	71.0	54,966	42.5	39.2	15,013	11.4
Sept98_5	75.2	196,403	277.2	37.7	43,342	59.3

It is hoped that using the recombination algorithm will allow warm started solves that are quicker than random start solves, and have a better chance of finding the global minimum. The table above compares solves with a random start and with a warm start from a cluster combination, averaged over ten solves. For all three problems the warm started solve was about four times quicker than the random start solve. The reduction in time is roughly proportional to the reduction in LP iterations.

For both Sonoco97_1 and Sonoco97_5 using the recombination algorithm with 20 random start then 30 warm started solves found the global minimum. This suggests that the recombination algorithm is better than just doing random starts, where the global minimum was found just twice and once in 200 runs of Sonoco97_1 and Sonoco97_5 respectively.

We now consider the larger problem Sept98_5, and how many random start solves should be done for this problem before recombining solutions. As an example we assume there is a total of around 4000 seconds available.

Using random starts with this time allowance about 15 solves can be done. In the table below we give the likelihood of the best local minimum found being within a given range of the global minimum, using information from a sample of 200 random start solves. Sept98_5 has a huge number of local minima (see Figure 5.7), therefore for presentation purposes we group solves in bands by objective value. The different bands are for objective values less than 0.001% above the global minimum, between 0.001% and 0.01% above the global minimum, between 0.01% and 0.1% above the global minimum, and above 0.1% above the global minimum.

	<0.001%	0.001%-0.01%	0.01%-0.1%	>0.1%
Random starts	0.000	0.072	0.294	0.633

Using the recombination algorithm a good use of 4000 seconds is to perform a cycle of just four random start then four warm start solves, then repeat this cycle twice. The table below shows that the probability of finding good local minima is greatly increased.

	<0.001%	0.001%-0.01%	0.01%-0.1%	>0.1%
Recombination algorithm	0.271	0.217	0.485	0.027

6.4 Summary

In this chapter we described the multifactory problem and showed it can be decomposed as either a row linked or column linked problem. Three decomposition methods were tested. Benders decomposition could solve only small test problems. The dual cutting plane method could solve larger problems, but was only competitive in number of outer iterations on multifactory problems with simple transport problems.

Augmented Lagrangian was shown to have theoretical advantages over standard Lagrangian, in coping with nonconvexity and degeneracy. We developed an augmented Lagrangian algorithm AAL1 which used frequent updating of $\boldsymbol{\lambda}$ by a new update formula, and used piecewise linear sections to approximate the quadratic penalty. For most problems this algorithm required fewer outer iterations than the dual cutting plane method. For the large P_e test problem AAL1 required less estimated work than a direct SLP solve.

We developed two algorithms to find the global minimum of multifactory problems, both exploiting the near separability of the problem. The first was able to find the global minimum

exactly for small test problems. The second was shown to be superior to performing repeated random start solves for finding the global minimum of larger test problems.

6.4.1 Further work

As it is likely an effective algorithm will be to some extent tailored to the problem classes encountered, it is essential to test the proposed dual cutting plane and augmented Lagrangian algorithms on more test problems, as they arise. The algorithms can then be improved and eventually used to solve real problems.

Chapter 7

Silo pooling problem

Sloten International is a company in the Netherlands which makes milk replacement fluid for young animals. Their factory can be modelled by a *silo pooling problem*. Silo pooling problems, or simply silo problems, are a new type of pooling problem that we introduce in this chapter. They are generalised pooling problem that includes *silos*. The presence of silos adds integer constraints to the already nonlinear factories and so makes them MINLPs.

In Section 7.1 we introduce the the Silo model. In Section 7.2 we introduce a new algorithm to solve such MINLPs by decomposing them into NLP and MIP subproblems. In Section 7.3 we consider two possible extensions to the model, the *silo grouping model* and the *unrestricted model*. These are both modelled and approximately solved.

7.1 Introduction

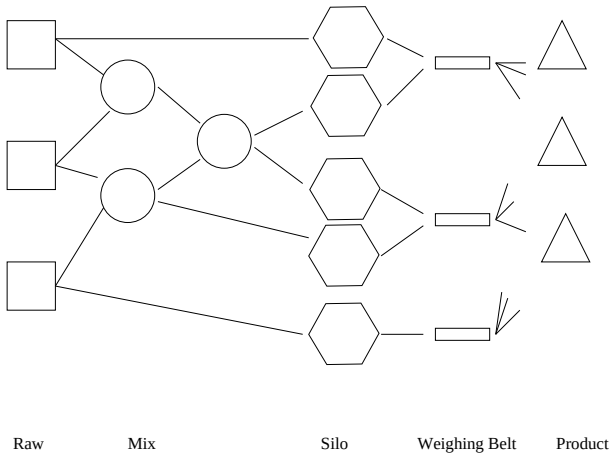


Figure 7.1: Example silo problem Ex1

Figure 7.1 shows an example silo problem, based on the GPP Factory3 shown first in Figure 4.5. Throughout this chapter Ex1 will be used as an example silo factory. The problem

features raws, mixes and products, all of which were previously introduced in the description of the pooling problem in Chapter 4. Because of the mixing that occurs in the silo problems they are nonlinear problems. The problem also features *silos* and *weighing belts*. The silos are bins which are fed from raws and mixes, and feed into the products, via the weighing belts. Each silo has a single fixed input, of either a raw or a mix. We call this the *silo allocation*. We call the arrangement of the silos into groups the *silo groupings*. During the production of each product each weighing belt can be attached to only one of the silos in each silo group, which implies that each product can only be fed by one silo from each group. We call the choice of silos for each product the *product allocation*. To enforce the conditions on product allocations the model needs variables that can only take integer values. We call the constraints restricting variables to be integer *integer constraints*. The standard version of the problem is the one described above. In it the silo groupings and the silo allocations are fixed. More complicated versions of the problem, where silo groupings and silo allocations can be varied, are considered in Section 7.3.

In Figure 7.1 the silos are explicitly shown as separate bins from the raws and mixes. It is possible that, in some factories, the weighing belts are applied directly to the raws and mixing bins, and there are no physical silos. However, we always model silo problems with an explicit layer of silo bins, whether or not one really exists in the factory. Silos are added specifically as inputs to the products. Each silo has just one fixed input, so modelling in this way is not likely to make the model any harder to solve. The advantage of modelling all problems like this is it means the same representation can be applied to all silo problems.

The weighing belts are not included in the actual model, and are drawn in Figure 7.1 just to highlight the silo groupings. Note that the bottom weighing belt has only one input. A silo group consisting of only one silo like this does not cause any integer constraints. The input to the silo, in this case a raw, can be freely chosen by all the products.

In the remainder of this introduction we give the full MINLP model of the the silo problem, look at possible methods to solve MINLPs, then introduce the particular silo problems known as the Sloten problems.

7.1.1 Mathematical model

Each silo problem is a GPP, with the addition of silos. We therefore model it as far as possible using our extended q-formulation for pooling problems developed in Chapter 4. Table 7.1 gives the structural parameters of the silo model.

R, M, S, G, P, N	number of raws, mixes, silos, silo groups, products and nutrients
G_g^{size}	number of silos in silo group g
G_g^{start}	position of first silo in silo group g

Table 7.1: Structural parameters of the silo model

X_{rn}	of nutrient concentration n in raw r
y_{mn}	nutrient concentration n in mix m
h_{sn}	nutrient concentration n in silo s
z_{pn}	nutrient concentration n in product p
C_r	cost of raw r per unit
e_s	cost of silo s per unit
f_p	cost of product p per unit
t_r	mass of raw t
u_m	mass of mix m
w_s	mass of silo s
V_p	mass of product p to be made
α_{mr}	proportion of mix m input direct from raw r
$\beta_{m'm}$	proportion of mix m' input direct from mix m
θ^R_{sr}	binary indicator equal to 1 if silo s has input of raw r, else 0
θ^M_{sm}	binary indicator equal to 1 if silo s has input of mix m, else 0
κ_{pr}	proportion of product p input direct from raw r
ζ_{ps}	proportion of product p input direct from silo s
ζ^I_{ps}	upper limit on proportion ζ_{ps} (binary variable)
q_{sr}	proportion of silo s input direct or indirect from raw r

Table 7.2: Parameters (upper case) and variables (lower case) of the silo model

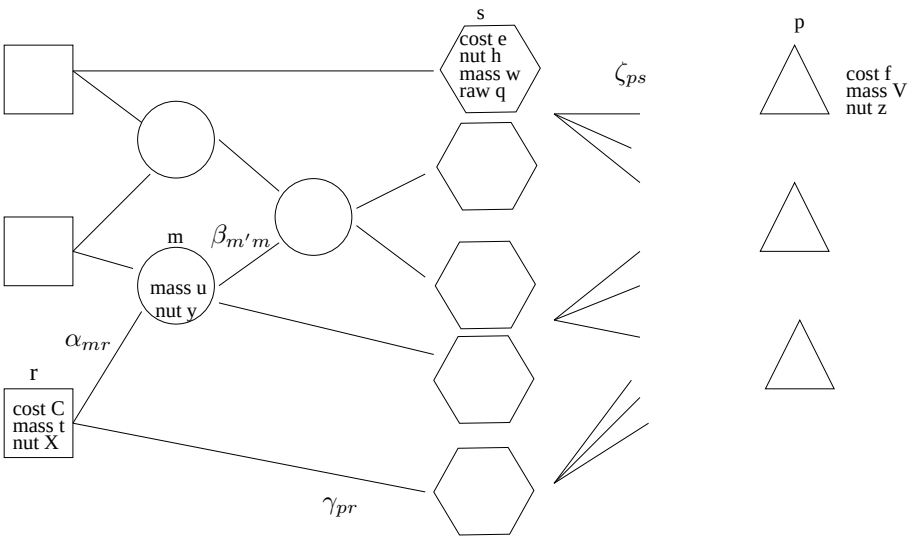


Figure 7.2: Silo problem Ex1 fully labelled

Table 7.2 gives the parameters and variables used in our model, and Figure 7.2 shows a labelled example silo problem. As in the model of the GPP, for simplicity we define variables for all possible flow between bins, even though in reality many of these flow proportion variables will be zero. We also do not explicitly show the weighing belts here in the factory here, instead they will be implicitly modelled by bounds on $\boldsymbol{\zeta}$.

Concerning the flow proportion variables flow into silos does not require a variable, as each silo has only one input. The parameters $\boldsymbol{\theta^R}$ and $\boldsymbol{\theta^M}$ are used to indicate which is the input to each silo. With the introduction of silos there is no direct flow from raws or mixes to products. However, in the case of a single silo, like the bottom one in Ex2, there are no limitations on the input of that silo going to products. Thus we consider the flow going direct from the silo

input, here a raw, to the product, and use the label κ_{pr}. This is similar to the variable for flow proportion to a product from a raw of γ_{pr} used in the GPP model. In real problems the direct flow to products is always from adding of a few extra raw ingredients (not mixes) into a product, so we only need to consider direct flow to products as coming from raws.

We state the complete silo model below in (7.1)-(7.11), omitting the simple bounds on variables. We think of constraints (7.2)-(7.6), which describe how the silos are made out of the raws and mixes, as being in the first part of the model, and constraints (7.7)-(7.11), which describe how the silos are made out of products, as being in the second part of the model.

$$\min \sum_{r=1}^{R} C_r t_r \tag{7.1}$$

$$\text{s.t.} \sum_{r=1}^{R} \alpha_{mr} X_{rn} + \sum_{m=1}^{M} \beta_{m'm} y_{mn} = y_{mn} \qquad m' \in 1 \dots M,\ n \in 1 \dots N, \tag{7.2}$$

$$\sum_{r=1}^{R} \alpha_{m'r} + \sum_{m=1}^{M} \beta_{m'm} = 1 \qquad m' \in 1 \dots M, \tag{7.3}$$

$$\sum_{r=1}^{R} \theta^R_{sr} X_{rn} + \sum_{m=1}^{M} \theta^M_{sm} y_{mn} = h_{sn} \qquad s \in 1 \dots S,\ n \in 1 \dots N, \tag{7.4}$$

$$\sum_{m=1}^{M} \alpha_{mr} u_m + \sum_{s=1}^{S} \theta^R_{sr} w_s = t_r \qquad r \in 1 \dots R, \tag{7.5}$$

$$\sum_{m'=1}^{M} \beta_{m'm} u_m + \sum_{m=1}^{M} \theta^M_{sm} w_s = u_m \qquad m \in 1 \dots M. \tag{7.6}$$

The objective and constraints here describe the manufacture of silos from raws and mixes. This is therefore fundamentally the same model as the GPP model for maxing products out of raws and mixes, introduced in (4.5)-(4.11). The model here is slightly simpler though, as each silo has one fixed input.

The objective (7.1) is to minimise the total cost, which is the sum of raw cost per unit times raw mass. Constraint (7.2) defines the nutrient concentration in each mix, by considering the flow into that mix from raws and the flow from other mixes. Constraint (7.3) says that the sum of proportional flows into each mix is one. Constraint (7.4) defines the nutrient concentration in each silo. The nutrient concentration in each silo is the same as the nutrient concentration in whichever raw or mix supplies the silo (only one value of the binary parameter θ^R_{sr} and θ^M_{sm} is nonzero for each silo s). Constraint (7.5) defines the mass of each raw, by considering the flow of that raw into mixes and products. Constraint (7.6) defines the mass of each mix, by considering the flow of that mix into other mixes and into products.

$$\sum_{r=1}^{R} X_{rn}\kappa_{pr} + \sum_{s=1}^{S} h_{sn}\zeta_{ps} = z_{pn} \qquad p \in 1 \dots P,\ n \in 1 \dots N, \tag{7.7}$$

$$\sum_{r=1}^{R} \kappa_{pr} + \sum_{s=1}^{S} \zeta_{ps} = 1 \qquad p \in 1 \dots P, \tag{7.8}$$

$$\sum_{p=1}^{P} V_p\zeta_{ps} = w_s \qquad s \in 1 \dots S. \tag{7.9}$$

Constraints (7.7)-(7.9) describe the production of products from silos. Constraint (7.7) defines the product nutrient concentration $\boldsymbol{z}$. Constraint (7.8) is a convexity constraint on the silo input and Constraint (7.9) defines the silo mass $\boldsymbol{w}$.

$$\sum_{s=G_g^{start}}^{G_g^{start}+G_g^{size}-1} \zeta_{ps}^I \leq 1 \qquad g \in 1 \dots G,\ p \in 1 \dots P, \tag{7.10}$$

$$\zeta_{ps} \leq \zeta_{ps}^I \qquad p \in 1 \dots P,\ s \in 1 \dots S. \tag{7.11}$$

Constraints (7.10)-(7.11) enforce the integer conditions. Recall that $\boldsymbol{\zeta^I}$ is a binary variable. Constraint (7.10) says that only one silo per silo group can feed into each product. Constraint (7.11) bounds the flow $\boldsymbol{\zeta}$ from silos to products to be less than $\boldsymbol{\zeta^I}$.

We refer to the silo model above as the *standard silo model*. Later we will introduce the *relaxed standard silo model* of (7.1)-(7.9). This is the standard silo model without the inclusion of the two integer constraints above.

7.1.2 MINLP

The silo problem is a MINLP. A survey of methods for solving MINLPs can be found in Abramson [2002], Leyffer and Linderoth [2005] or Letchford [2009]. We give only a brief overview here, focusing on methods that are relevant to the new solve methods we will introduce later, which rely on the near separability of the silo MINLP into NLP and ILP parts.

Some MINLPs can be represented as *bilinear programs*, where the variables can be partitioned into two subsets, such that when each set is fixed the remaining problem is comparatively easy to solve. Here the partition would make the MINLP into an NLP and ILP. Yuceer and Berber [2004] approximately solve a MINLP by heuristically fixing a subset of variables to make the problem an ILP, which is then solved. El-Samahy [2006] approximately solves a MINLP by first tentatively fixing all the integer variables to make the problem an NLP. He solves this, then heuristically adjusts some of the values of the integer variables.

An interesting decomposition algorithm to solve ILPs is the feasibility pump algorithm of Fischetti et al. [2005]. Although this is a method to solve ILPs, it is relevant to the solver

ASR1 that we will introduce later to solve MINLPs. Feasibility pump can be viewed as a decomposition of an ILP into the integer constraints and the noninteger constraints. In the following description of their algorithm we use *integer feasible* to refer to a point that satisfies all the integer constraints, and *constraint feasible* to refer to a point that satisfies the noninteger constraints. The aim of the algorithm is to find a point which is both integer and constraint feasible, or at least find a point which is close to feasible, that can then be used as a warm start point for the original ILP solve.

1	Find a constraint feasible point $\boldsymbol{x}_1^*$.
2	Round this to nearest integer point, $\boldsymbol{x}_1^I$.
3	Solve an LP to find constraint feasible point $\boldsymbol{x}_2^*$.
4	Continue alternating between constraint feasible but integer infeasible points $\boldsymbol{x}_k^*$, and integer feasible but constraint infeasible points $\boldsymbol{x}_k^I$.

Table 7.3: Feasibility pump algorithm

An outline of the algorithm is given above. In Step 1 the algorithm is initialised by solving the LP formed from relaxing the integer constraints in the ILP, to give a solution point that is constraint feasible but typically integer infeasible. Let this point be called $\boldsymbol{x}_1^*$. In Step 2 $\boldsymbol{x}_1^*$ is rounded to the nearest integer feasible point, called $\boldsymbol{x}_1^I$, at the expense of constraint feasibility. In Step 3 the integer constraints are again relaxed and point $\boldsymbol{x}_2^*$ is found by solving an LP to find the closest constraint feasible point to $\boldsymbol{x}_1^I$, which is then rounded to form $\boldsymbol{x}_2^I$. This method continues indefinitely in Step 4 until a satisfactory point is reached.

In Fischetti and Lodi [2005] they consider how the first constraint feasible point, $\boldsymbol{x}_1^*$, is found in Step 1. This can be solving an LP that is relaxed by augmenting each constraint with an artificial variable multiplied by a binary variable, that measures the constraint violation. The new objective is simply the sum of the binary variables. If any constraint cannot be satisfied with just the original variables the artificial variable in that constraint, hence the corresponding binary variable, becomes nonzero, and the objective increases by one. The addition of the binary variables turns this LP (back) into an ILP. This ILP is solved using another feasibility pump, until all integer constraints are satisfied. The artificial and binary variables are then dropped and the resulting LP solved to find the point $\boldsymbol{x}_1^*$, and the feasibility pump continues.

Some MINLPs can be decomposed into NLPs and ILPs, as is done in ASR1. If this is possible the NLP can be solved by any of the methods for solving NLPs detailed in Chapter 5. Several methods to solve ILPs are discussed in Lee [2007]. The most common (global) method is *branch and bound*, which works by efficiently enumerating the solutions of many (but far from all) of the LPs that result from fixing the integer variables. These LPs can be *strengthened* by various *cuts* that to some extent replicate the integer constraints. Branch and bound is a global method, meaning it is guaranteed to find the global minimum of an ILP. Versions of branch and bound, or the similar method of branch and cut, or branch and price, can also be applied directly to MINLP problems. If each NLP is solved to global optimality these methods can find

the global minimum of the MINLP. We consider the solve of the ILP formed during algorithm ASR1 in Chapter 7.2.2.

Because of their difficulty MINLPs are often solved by heuristic methods. These methods try to generate good solutions quickly, without a guarantee of success. Heuristic methods for MINLP include scatter search, simulated annealing, genetic and evolutionary algorithms, pattern search and tabu search. These are all described in Abramson [2002]. Several of the methods, such as scatter search, are extensions of methods developed to solve NLPs. An example scatter search algorithm for MINLPs is Ugray et al. [2007], which repeatedly fixes the integer variables at randomly generated values, then solves the resulting NLP from the most promising integer assignments. Often heuristic methods are complemented by *local improvement methods*. These attempt to refine the solution found, or at least show its local optimality, typically by attempting to change small parts of the solution.

By converting integer constraints to nonlinear constraints an MINLP becomes an NLP. As an example, the constraint forcing exactly one of the binary variables x_1 and x_2 to be nonzero can be converted to:

$$\begin{aligned} x_1 x_2 &= \mu, \\ x_1 + x_2 &= 1, \\ 0 \leq x_1, x_2 &\leq 1, \end{aligned}$$

where x_1 and x_2 are now continuous variables, and μ a parameter of how exactly the integrality must be enforced, with strict integrality when $\mu = 0$. The new nonlinear constraint is regrettably highly concave.

For solving specific MINLPs heuristics that exploit knowledge of the problem are often used. Current (2008) attempts by Sloten International to solve silo problems rely on the relaxed solution being close to the optimal solution. Sloten International solve the relaxed model (an NLP) by SLP using Integra_1, then manually allocate silos to each product to try and minimise *weighing belt conflicts*. A weighing belt conflict occurs when constraint (7.10) is violated, i.e. when for a particular product one weighing belt feeds the product more than one silo from its silo group. This manual allocation of silos may not be able to find a product allocation that has no weighing belt conflicts, indeed one might not exist for the given silo compositions. Furthermore, there is a risk that after the manual allocation the products will collectively use more than the upper bound on mass for some raws.

	R	M	S	G	P	N
Ex2	11	3	6	3	8	5
Sloten1	132	12	26	6	126	73
Sloten5	128	11	31	8	101	69

Table 7.4: Silo problems

7.1.3 Test problems

Table 7.4 gives the dimensions of the problems solved in this chapter. These are all new problems. For each problem we give the number of raws, mixes, silos, silo groups, products and nutrients. When counting the number of silos and silo groups we consider only *nontrivial silo groups*, meaning those groups with more than one silo in them. Trivial silo groups automatically have the weighing belt allocated to the only silo, so do not require integer constraints.

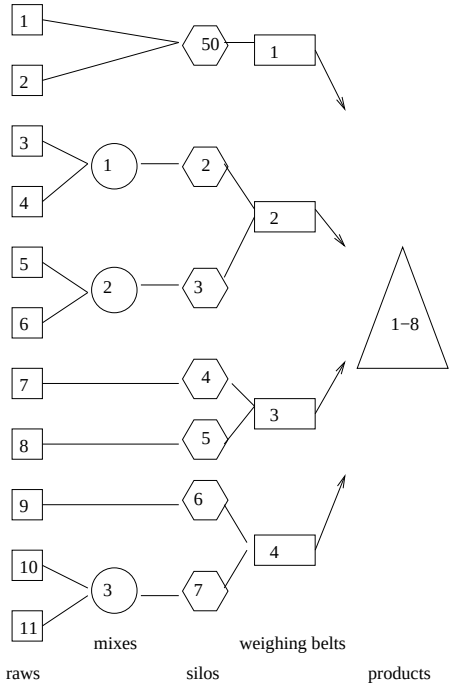


Figure 7.3: Ex2 silo problem

Figure 7.3 shows the small example silo problem named Ex2. We now look in more detail at the real problems Sloten1 and Sloten5.

Sloten factory

Figure 7.4 shows a schematic of the Sloten factory, located in the town of Sloten in the Netherlands. Details such as the actual size of the plant, how frequently it is optimised, and the true costs involved are not known to us.

Although we model and solve it as one factory, Figure 7.4 shows that it is actually made up of two linked factories. In the preliminary factory some raws are blended into mixes, which are then used in the main factory along with other raws to make the products. Three sets of raws

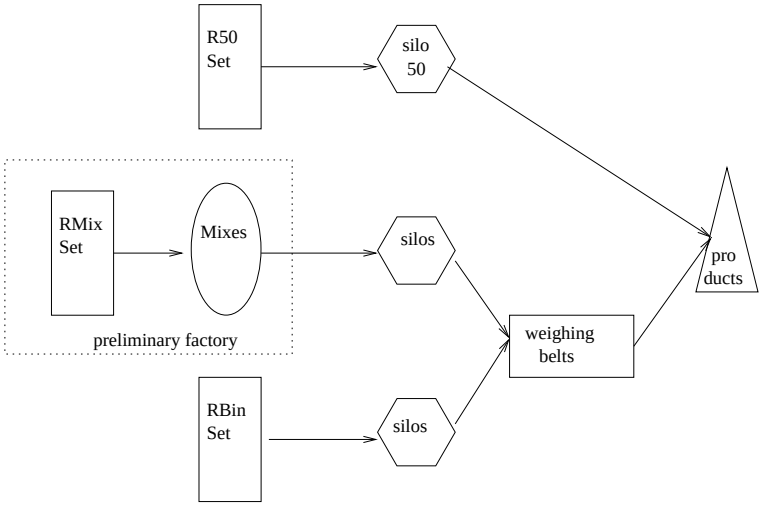
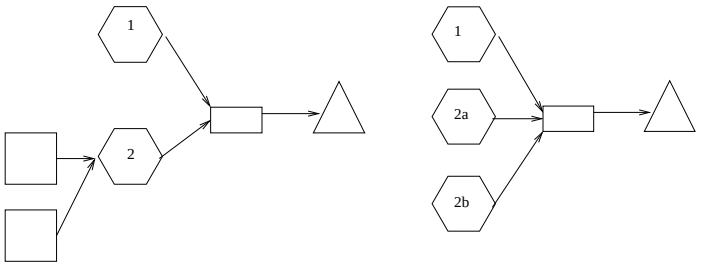


Figure 7.4: Sloten factory

are shown. Raws in $RMixSet$ are mostly fats. They form the mixes in the preliminary factory. Raws in $R50Set$ are mostly ingredients used in small quantities, such as nutrient supplements. These raws all feed into the products via Silo 50. Silo 50 is unlike the other silos. Multiple raws can feed into it, become mixed and feed into the products, and a different combination can be chosen for each product. There is no mass limit and no weighing belt. This silo does not need integer constraints, so is considered a trivial silo group. Raws in $R50Set$ can be thought of as feeding directly into the products. Raws in $RBinSet$ feed into the products via several different silos. There are a small number of raws that appear in more than one set.



(a) (b)

Figure 7.5: Silo with multiple inputs (a) reformulated to standard form (b)

The Sloten factory has a few silos with multiple possible inputs, of which at most one can be used by each product. This is not the standard form, so we reformulate, by duplicating silos. A silo with k possible inputs is reformulated as k silos, each with one input. Figure 7.5(a) shows an example problem where Silo 2 has two possible inputs. Each product can select either Silo 1, or either of the inputs to Silo 2. This is reformulated to become Figure 7.5(b), where the three possible inputs to the weighing belt are now labelled as Silo 1, Silo 2a and Silo 2b.

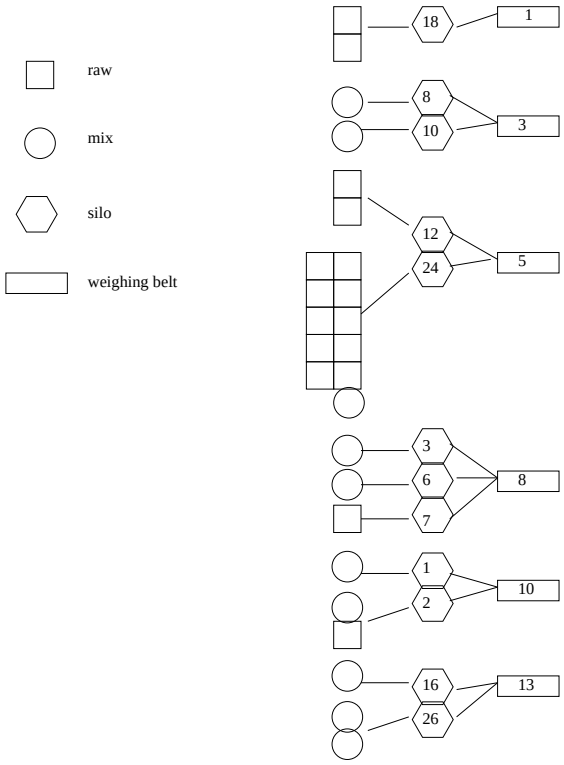


Figure 7.6: Sloten1 nontrivial silo groups

There are different versions of the Sloten problem, which have slightly different solve data. We look at two versions called Sloten1 and Sloten5. Figure 7.6 shows the nontrivial silo groups and weighing belts, and the inputs to the silos, for Sloten 1. When solving this problem the silos with multiple inputs are reformulated to give multiple equivalent silos (see Figure 7.5), so for example there are in total 13 possible inputs to the products from weighing belt five. The six silo groups have sizes $\{2, 2, 13, 3, 3, 3\}$.

Sloten5 is a harder version of the problem. There is more flow possible between bins, meaning there is more nonlinearity. Furthermore there are more silo groups, hence more integer constraints. The eight nontrivial silo groups have sizes $\{2, 3, 13, 2, 3, 3, 2, 3\}$.

Weighing belt limits

When it was first described to use the Sloten problem had the additional feature of *weighing belt limits*, which have since been dropped. Weighing belt limits are bounds on the flow proportion to each product from each weighing belt. Typical weighing belt limits were something like 4%-80%, meaning each product could only get between 4% and 80% of its mass from the selected silo (whichever it was) on that weighing belt. The size of the lower and upper limits came from comparing the physical speed restrictions of the weighing belt with the speed that the products can be produced, and varied slightly between belts, but always had the twenty times ratio between lower and upper limit. An upper limit of less than 100% meant that no product

could get all its mass from one weighing belt. A lower limit above 0% meant that each silo must be used either 0%, if not selected, or above some minimum threshold, if selected. This discontinuity in allowable percentage use of silos required extra integer variables in the model.

In an early version of the Sloten problem two particular products were linked to only one silo each. The given weighing belt limits gave an upper bound of flow proportion to the products from any one silo of just 80%, hence the problem was infeasible. In practice 100% flow proportion to those products was achieved by slowing the physical speed at which the product was produced. By a similar change of relative production speeds the lower bounds on flow proportion could be set to 0%. These changes in relative production speed are always possible, hence the weighing belt limits were removed.

In preliminary trials we solved some versions of the Sloten problem with and without the weighing belt limits. It was found that the weighing belt limits hardly affected the problem difficulty, or optimal objective value. This may be at least partially because the problems had been designed with flow proportion bounds that in many cases implicitly forced satisfaction of the weighing belt limits. In the remainder of the chapter there is no further discussion of weighing belt limits, which we assume have been permanently removed.

7.2 Decomposition

It was mentioned above that the first part of the model is determining the silo compositions, and the second part is determining the product allocations. We now formally define these two parts of the problem. This is a novel decomposition that we will later use to solve the silo model.

Part 1

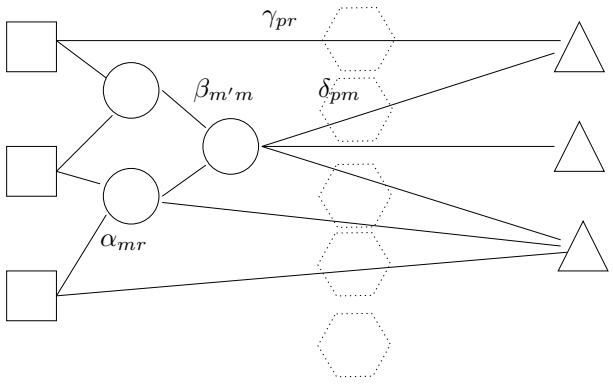


Figure 7.7: Part 1 of silo problem Ex1

In Part 1 we drop the silos and weighing belts, and simply make products directly out of raws and mixes. Figure 7.2 shows the first part of the problem Ex1. The silos and weighing belts are not included. This problem is a standard GPP problem, and so has the standard GPP

constraints of (4.5)-(4.11). Note that in Figure 7.2 flow is only possible from some mixes and raws to some products. Which raws and mixes are accessible to each product can be controlled in a GPP by changing the simple bounds on variables $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$.

We gave the full standard silo model above in (7.1)-(7.11). The Part 1 solve plays the role of determining the silo compositions, so is like the constraints (7.2)-(7.6). Constraints (7.2)-(7.3) are for supply to mixes. These are explicitly included in the Part 2 model. Constraints (7.4)-(7.6) define the silo nutrient concentrations and define the raw and mix masses in terms of silo masses. These are implicitly modelled by the Part 2 model with a similar trio of constraints that refer to the products instead of the silos.

Part 2

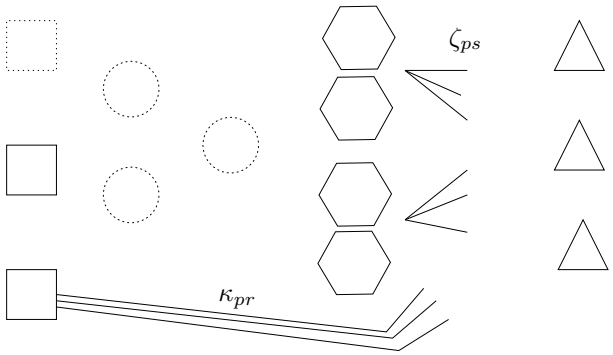


Figure 7.8: Part 2 of silo problem Ex1

Figure 7.2 shows the Part 2 model. Part 2 is just the problem of choosing product allocations, so omits the whole supply network to the silos, and consists of just the product inputs. These inputs are the silos, through flow $\boldsymbol{\zeta}$ and the associated integer constraints, and raws that can go direct to products, with flow $\boldsymbol{\kappa}$. Thus as well as the silos we include the raws that flow directly to products. We must also include any raws in the model that have mass bounds. This is because the product allocations could end up using a combination of silos in a manner that violates raw mass bounds. However, not all raws need to be included in the model, only those with mass bounds. This is indicated in Figure 7.2 by showing one of the raws with a dotted outline, meaning it is not included in the Part 2 model.

$$\min \sum_{p=1}^{P} V_p f_p \tag{7.12}$$

$$\text{s.t.} \sum_{r=1}^{R} X_{rn}\kappa_{pr} + \sum_{s=1}^{S} h_{sn}\zeta_{ps} = z_{pn} \qquad p \in 1 \ldots P,\ n \in 1 \ldots N, \tag{7.13}$$

$$\sum_{r=1}^{R} \kappa_{pr} + \sum_{s=1}^{S} \zeta_{ps} = 1 \qquad p \in 1 \ldots P, \tag{7.14}$$

$$\sum_{p=1}^{P} V_p \zeta_{ps} = w_s \qquad s \in 1 \ldots S, \tag{7.15}$$

$$\sum_{s=G_g^{start}}^{G_g^{start}+G_g^{size}-1} \zeta_{ps}^I \leq 1 \qquad g \in 1 \ldots G,\ p \in 1 \ldots P, \tag{7.16}$$

$$\zeta_{ps} \leq \zeta_{ps}^I \qquad p \in 1 \ldots P,\ s \in 1 \ldots S, \tag{7.17}$$

$$\sum_{r=1}^{R} C_r \kappa_{pr} + \sum_{s=1}^{S} f_p \zeta_{ps} = e_s \qquad p \in 1 \ldots P, \tag{7.18}$$

$$\sum_{p=1}^{P} V_p \kappa_{pr} + \sum_{s=1}^{S} q_{sr} w_s = t_r \qquad r \in 1 \ldots R. \tag{7.19}$$

The full Part 2 model is given above, excluding the simple bounds on each variable. The objective (7.12) is again to minimise the cost, defined here by the total cost of the products. The reason for using this objective, rather than the standard objective of raw cost times mass, is because we are not calculating all of the raw masses here, only the masses of those raws which have mass limits. Constraints (7.13)-(7.14) are for the product nutrient concentrations. Constraint (7.15) calculates the mass of each silo, for use in constraint (7.19). Constraints (7.16) and (7.17) are the integer constraints. All of these first five constraints are also in the full silo model, as (7.7)-(7.11). There are two further constraints in the Part 2 model. Constraint (7.18) defines the price of each product (indirectly), for use in the objective function. Constraint (7.19) defines the masses of the raws, to check that they are within bounds.

Because the silo compositions are taken as fixed in Part 2, the variables for silo nutrient concentration, $\boldsymbol{h}$, silo cost per unit, $\boldsymbol{e}$ and of silo input direct or indirect from raws $\boldsymbol{q}$, are all taken as fixed. Thus the Part 2 model (7.12)-(7.19 is linear. Because of the integer constraints it is an ILP.

We now develop a heuristic algorithm, ASR1, that solves standard silo problems. It works by decomposing the MINLP into the NLP of Part 1 and the ILP of Part 2. In Section 7.2.1 we describe the algorithm. In Section 7.2.2 we look in more detail at solving the ILPs. In Section 7.2.3 we give results for solving silo problems with ASR1.

7.2.1 Algorithm ASR1

Algorithm ASR1 is a new algorithm designed for the standard silo model. It is based on the decomposition into Part 1 and Part 2. The Part 1 solve can be forced to follow a particular product allocation, by imposing bounds on the flow proportions into products. So if Part 1 is given a feasible product allocation, it can then be solved to try and improve the objective by changing the flow into the silos, while retaining the given product allocation. Similarly the Part 2 solve can be forced to obey given silo compositions, by having the nutrient concentration, price and proportions of each raw in each silo fixed. So if Part 2 is given feasible silo compositions it can be solved to improve the objective by changing the product allocations, while retaining the given silo compositions

1	Solve Part 1 NLP.
2	For the solution calculate silo compositions $\hat{\boldsymbol{h}}$, silo costs $\boldsymbol{w}$ and proportion of each raw in each silo $\boldsymbol{q}$. Fix these three variables at their current values.
3	Solve Part 2 ILP.
4	For the solution calculate maximum flows from products to raws and mixes $\overline{\boldsymbol{\gamma}}$ and $\overline{\boldsymbol{\delta}}$. Fix these bounds.

Table 7.5: One iteration of a solve of the silo problem by algorithm ASR1

Table 7.5 gives an outline of an iteration of ASR1. We assume for now that each NLP and ILP solve is feasible. The possibility of infeasibility, and how to start each iteration, are discussed in the subsections below.

In Step 1 we solve Part 1. At the solution we have an objective value, and a value for the mix compositions. As each silo has a unique input of either one raw or one mix, once the mix compositions are known we know what the silo compositions must be. In Step 2 we fix these silo compositions. Given the flow in the network we can also calculate the price of each silo, and the amount of each raw directly or indirectly in each silo. These are also fixed.

In Step 3 we solve the Part 2 ILP. At the solution we get a new objective value, and a value for the flow proportion variables between the silos and products. These flows limit each product to receive input from at most one silo per silo group. In Step 4 we fix these product inputs. As each silo has a unique input, limiting the flow from silos to products must also limit the direct flow from any raws or mixes to products. We enforce this by bounding the $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ variables to be zero where there is no flow allowed. Once these flows are bounded we know that the next solve of Part 1 will continue to respect the current product allocations.

The algorithm is a coordinate axes search, alternately adjusting mix compositions in Part 1 and product allocations in Part 2. In each part 1 solve the product allocations are fixed, and the solution reduces the objective value while retaining an integer feasible solution. The result of a Part 1 solve is new mix compositions, hence silo compositions, for Part 2. In each Part 2 solve the silo compositions are fixed, and the objective value is reduced while retaining

feasible silo compositions. The result of a Part 2 solve is new product allocations, hence new flow bounds for Part 1.

Convergence to a local minimum occurs when either part fails to produce a new solution point. In practice this usually occurs because the ILP does not modify the product allocation, rather than the NLP not modifying the mix composition, as product allocation is discrete and mix composition is continuous. Once a local minima is found it is common in MINLP algorithms to try to improve on the solution, by local improvement methods. Because ASR1 is a coordinate axes search it is possible that the local minimum found could be improved on by simultaneously changing both axes, i.e. simultaneously changing the mix composition and the product allocation. Although local improvement methods were found to be effective for solving versions of the small test problem Ex2, for the large Sloten problems it was more effective to simply perform multiple random starts. There are therefore no local improvement methods in ASR1.

Feasibility

We noted above that, given a feasible solution, solving each part will retain feasibility. To ensure that we can still proceed when either part is not feasible we relax the simple bounds on the product nutrient concentration $\boldsymbol{z}$, by the addition of artificial variables. This technique is similar to the one used to start the feasibility pump algorithm, discussed above, and is also similar to the relaxation in SLP phase one, shown in (5.14). We add nonnegative artificial variables $\boldsymbol{z^L}$ and $\boldsymbol{z^U}$, that relax the original product nutrient constraints $\underline{\boldsymbol{z}} \leq \boldsymbol{z} \leq \overline{\boldsymbol{z}}$:

$$\begin{aligned} \underline{\boldsymbol{z}} \leq \boldsymbol{z} + \boldsymbol{z^L} - \boldsymbol{z^U} \leq \overline{\boldsymbol{z}}, \\ \boldsymbol{z^L}, \boldsymbol{z^U} \geq 0. \end{aligned} \tag{7.20}$$

We add penalties in the objective for the artificial variables being nonzero. In both parts the objective value is now the total cost of the factory, plus the penalties on artificial variables. The penalties are sufficiently high that the artificial variables are only nonzero when there is no other way of getting feasibility. This is reminiscent of the *big M penalty method* sometimes used in the simplex method, which prioritises minimising infeasibility while also minimising objective value. This is in contrast to the feasibility pump algorithm, where the original costs are discarded. Each solve of either part will reduce the artificial penalty if possible. Once the penalty is zero, it will remain zero, and a feasible solution will be retained.

It is theoretically possible that ASR1 can fail to find a local minimum when one exists. All ILPs can always be solved if the products can have any nutrition concentration. However, it is possible that an NLP will be infeasible even with the relaxed product nutrient constraint. This can happen if the product allocations lead to mix conditions which cannot be satisfied even with

use of the artificial variables. It is also possible that ASR1 can fail to converge if different local minima are found each NLP solve. In testing though neither of these two problem situations ever arose, and ASR1 always terminated with a local minimum.

Multiple solves

ASR1 is only a local solver, so is not guaranteed to find the global minimum. To increase the chance of finding the global minimum we can perform multiple starts from different points. In terms of notation we talk about doing several *solves* in one *run* of a problem. We now consider how to generate different random start points for each solve.

If just one solve is required we can simply begin in Step 1 by solving a Part 1 model with no restriction on the flows. This is a solve of the relaxed model, which consists of (7.1)-(7.9). This solve gives a lower bound on the final objective value (assuming the NLP finds the same local minimum each iteration). For other solves in the run we must find other random start points. To do this we solve a preliminary NLP with random costs on the raws, then solve a preliminary ILP to find the best product allocation for these random costs. These product allocations are then fixed in the first proper NLP solve. This gives a wide range of start points.

7.2.2 Solving ILP subproblem

ASR1 requires an NLP and ILP solver. As the NLPs are GPPs the Integra_1 solver of Chapter 5 can be used. This is an efficient algorithm to find local solutions.

We now consider the solve of the ILP. An ILP is a problem that is linear, but has the addition of integer constraints. Most solve methods exploit this similarity to LP to try and solve the ILP by solving a series of LPs. For certain classes of ILP specialised solvers exist (see Chapter 8). For general ILPs the most effective method to solve them is often *branch and bound*, discussed briefly above in Section 7.1.2.

In this section we apply the branch and bound solvers CPLEX and BAB to the silo ILP. CPLEX is a leading commercial solver, and BAB is a University of Edinburgh code that is only suitable for solving small problems. Because of the expense of commercial solvers there is interest from Sloten International in being able to solve MINLPs without ever having to use CPLEX. We therefore consider both solving the ILP directly, with CPLEX, and using the dual cutting plane method (introduced in Section 6.2.3) to create a series of smaller ILP problems of the size that BAB can solve. A further motivation for using the dual cutting plane method is that decomposition methods, such as this, can be superior for very large problem instances, which may need to be solved in the future.

Recall the ILP model consists of the objective and constraints (7.12)-(7.19). Of these only (7.15) and (7.19) link the products. We therefore propose decomposing the problem by product, relaxing these two constraints. As there are lots of products there will be a large number

of subproblems, each small ILPs consisting of the original ILP constraints with the relaxed constraints in the objective. For each subproblem rather than summing over p each constraint will be formed for just one value of p, call it $\hat{p}$. As constraint (7.19) is only formed for raws that have bounds, which may be only a small number of the raws, there may be few linking constraints. This is therefore an attractive decomposition, as we expect many subproblems with few linking constraints.

		LP iterations	B&B branches	time
Sloten5 ILP	CPLEX	2000	10	6
	BAB		fails	
Sloten5 ILP subproblem	CPLEX	14	0	0.02
	BAB	29	5	0.02

Table 7.6: Comparison of CPLEX and BAB

Table 7.6 compares CPLEX and BAB on the Sloten5 ILP and on one of the Sloten5 ILP subproblems resulting from decomposition (all the Sloten5 ILP subproblems are very similar). The Sloten5 ILP has about 27,000 variables of which about 3,000 are integer variables. It is solved by CPLEX in around 6 seconds, and does not solve by BAB. The CPLEX solve has 10 branch and bound branches, meaning in total just 11 LPs are solve. The quick solve of CPLEX is attributed to the small silo groups limiting the number of possible optimal integer allocations that need to be explored, hence limiting the number of branches. To illustrate this we attempted to solve a version of the Sloten5 ILP where the silo group sizes were increased to $\{6, 7, 2, 9, 3, 4, 2, 4, 2, 3, 2, 2\}$. CPLEX failed to find the solution after 600 seconds.

The Sloten5 ILP subproblems each have about 300 variables of which about 30 are integer variables. The table shows they solve easily using either CPLEX or BAB. The total time to solve an ILP by the dual cutting plane method is the time per subproblem solve, multiplied by the number of subproblems, multiplied by the number of dual cutting plane iterations. The solve of the Sloten5 ILP takes around 0.02 seconds per subproblem, has 101 subproblems, and requires about 30 iterations. Hence the total time is about 60 seconds, which is considerably slower than the 6 seconds required by the direct solve. However, it is possible that the dual cutting plane method could be sped up, by warm starting the subproblems. Also it is expected that for larger problems CPLEX may struggle, whereas the decomposition will scale well.

In this decomposition each subproblem is itself a (nonconvex) ILP. This means that it is very likely there is a duality gap. If a duality gap does occur the integer allocation found by the ILP may be suboptimal. A duality gap was observed on some Sloten problem versions, and meant that a lot of work had to be done after the decomposition to find a primal feasible solution. In ASR1 we therefore prefer the direct ILP solve by CPLEX.

7.2.3 Results for ASR1

We solved the three new silo problems using our algorithm ASR1; Ex2, Sloten1 and Sloten5. Note that the MINLP solvers Bonmin, FilMINT and MINLP-Fletcher were all able to (locally) solve Ex2, but not the Sloten problems.

Ex2

This small problem can be solved to local optimality by the MINLP code MINLP-Fletcher in a few seconds. ASR1 also solved this small problem easily. We actually solved many versions of it with various costs and bounds in the problem data, with these costs and bounds designed to create problems with relaxed model solutions that were very different from the integer feasible solutions. With multiple random starts ASR1 was still able to find the global minimum of each version, which typically had objective value about 20% above the best relaxed model solution.

Sloten1

The Sloten problems could not be solved by the MINLP code MINLP-Fletcher. Solving the relaxed problem NLP by SLP typically took around a minute. We will see that solving to find the integer solution by ASR1 took only a few times longer than this.

Each solve of Sloten1 with ASR1 took a small number of outer iterations, typically two to four. With multiple random starts the best solution value found was 1.887,349, just 0.002% above the best relaxed model solution found, which was 1,887,306.

iteration	Part 1 solution	Part 2 solution
1	1,887,306	1,887,351
2	1,887,349	1,887,349

The table above shows the output of the first solve of a run of solves of Sloten1 by ASR1. The significance of this being the first solve of the run is that the first Part 1 solve is simply the relaxed model. At the solution to this first Part 1 solve, 6 out of the 126 products have at least one infeasible product allocation, mostly due to conflicts on weighing belt five. This is the weighing belt with the most silos on it (recall a weighing belt conflict occurs when for a particular product one weighing belt feeds the product more than one silo from its silo group).

The first Part 2 solve has higher objective value than the first Part 1 solve, due to the addition of the integer constraints, then in the subsequent iteration the objective value falls. ASR1 terminates after the ILP solve fails to find a new integer allocation. The final objective value in this example solve was the best one found.

Sloten5

Each solve of this problem with ASR1 typically took between three and five iterations. With multiple random starts the best solution value found was 1,831,959, 0.279% above the best

relaxed model solution found, which was 1,826,863. Sloten5 is a harder problem to solve than Sloten1, which is reflected in the fact that each solve by ASR1 took more iterations, and there is a greater difference between the relaxed and integer feasible solution.

iteration	Part 1 solution	Part 2 solution
1	1,826,863	78,859,725
2	1,843,253	1,832,546
3	1,832,009	1,832,009

The table above shows the output of the first solve of a run of Sloten5 by ASR1. The first Part 2 solve has a very high objective value, as to get a feasible solution 27 of the products use artificial variables. The second Part 1 solve is feasible though and ASR1 quickly converges. The objective value in this example solve was above the best one found.

We may make Sloten5 infeasible by including Product 36, which is usually omitted. When this new problem version is solved by ASR1 the infeasibility is decreased each iteration, terminating with what is believed to be the global minimum of infeasibility, 0.04970. This example shows how ASR1 can be used to find the minimum of infeasibility for infeasible problems.

Overall effect of integer constraints

We now summarise how the addition of the integer constraints made the silo problem harder than a standard GPP.

Firstly, concerning objective value, we saw above that the objective values of the Sloten problems are very similar to the relaxed model solves. This indicates that the integer constraints have little effect on the solution. This may in part be because some of the nutrient concentration and flow proportion bounds given in the problem data are designed to implicitly force the solution to the relaxed model to be close to integer feasible.

Secondly, we consider the time to solve the MINLP once with ASR1, compared to the time to solve the relaxed model once. One solve by ASR1 includes a solve of the relaxed model, then a few further NLP and ILP solves. The relaxed model solve is actually the most expensive part of ASR1. The other NLP solves are all warm started, and the ILP solves are consistently fast when using a direct solve with CPLEX. The total time to solve using ASR1 is therefore typically only two or three times greater than the time to solve the relaxed model. The exact ratio of times depends on how fast the relaxed model solve is, which for the Sloten problems is highly sensitive to the solve parameters.

7.3 Model extensions

The model we have described and solved is the standard silo model. In it both the silo groupings and the silo allocations are fixed. We now consider two possible extensions to that model.

7.3.1 Changing silo groupings

We can form a new variant of the silo model by relaxing it such that the silo groupings can be changed, while keeping the group sizes and the number of weighing belts the same. Each silo still has a unique input. The advantage of changing the silo groupings is that it may allow better product allocations. For example, consider the three possible silo groupings for Ex1 shown in Figure 7.9. Perhaps with the original grouping of Figure 7.9(a) the objective value is hindered as many products want to use Silo 1 and Silo 2, but cannot. In the new groupings of Figure 7.9(b) or Figure 7.9(c) they can. Note that although they appear different silo groupings 7.9(b) and Figure 7.9(c) are functionally equivalent.

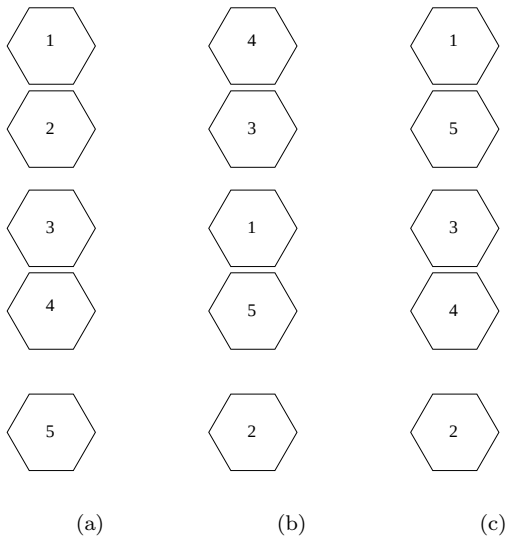


Figure 7.9: Original (a) and new (b),(c) silo groupings for Ex5

To properly model the ability to change the silo groupings involves forming the standard silo model and adding further integer constraints. Instead we use an approximate method here. We start with the relaxed model solution, which does not have any integer constraints so typically has several weighing belt conflicts, and try to adjust the silo groupings to make this solution integer feasible. We take the silo compositions as fixed and form what we call the *silo grouping ILP* to minimise weighing belt conflicts. This ILP is then solved by CPLEX.

Table 7.3.1 shows the parameters and variables of the model. The parameter $\boldsymbol{M}$ is a grid of binary variables that indicates which products want to use which silos. This determines how the silos should be grouped to avoid weighing belt conflicts. The variable $\boldsymbol{x}$ records where each silo moves to. We use the index s for the original silo position and s' for the new silo position, so for example if $x_{14} = 1$ it means the silo in original position 1 moves to position 4. The integer variable $\boldsymbol{y}$ records the resulting number of weighing belt conflicts. Note that $\boldsymbol{y}$ is not a binary variable, as the number of weighing belt conflicts for a given product and silo group can be greater than one.

S, P, G	number of silos, products, and silo groups
G_g^{size}	number of silos in silo group g
G_g^{start}	position of first silo in silo group g
M_{ps}	binary parameter that is equal to 1 if product p uses silo s in relaxed solution.
$x_{ss'}$	binary variable that is equal to 1 if the silo in position s moves to position s'.
y_{pg}	integer variable that records the number of weighing belt conflicts for product p in group g.

Table 7.7: Structural parameters and other parameters (all upper case) and variables (lower case) of silo grouping ILP

$$\min_{\boldsymbol{x},\boldsymbol{y}} \sum_{p=1}^{P} \sum_{g=1}^{G} y_{pg}, \tag{7.21}$$

$$\text{s.t.} \sum_{s'=1}^{S} x_{ss'} = 1 \qquad s \in 1 \dots S, \tag{7.22}$$

$$\sum_{s=1}^{S} x_{ss'} = 1 \qquad s' \in 1 \dots S, \tag{7.23}$$

$$\sum_{s=G_g^{start}}^{G_g^{start}+G_g^{size}-1} M_{ps} x_{ss'} \leq 1 + y_{pg} \qquad p \in 1 \dots P,\ g \in 1 \dots G. \tag{7.24}$$

The silo grouping ILP model is given in (7.21)-(7.24). There are also simple bounds limiting each variable to be nonnegative. The objective (7.21) measures the number of weighing belt conflicts. In a perfect solution this is zero. Constraint (7.22) ensures that each silo moves into exactly one position, and constraint (7.23) ensures that each position has exactly one silo moving into it. The left hand side of constraint (7.24) counts the number of silos used in each silo group by each product. If this is greater than one then on the right hand side the variable $\boldsymbol{y}$ must become nonzero and the objective increases.

	Nontrivial silo groups								total
Silos on belt	2	3	13	2	3	3	2	3	31
Weighing belt conflicts	9	0	18	0	5	3	0	0	35

We solve the silo grouping ILP for Sloten5. The table above gives information about the relaxed solution, showing for each of the eight nontrivial silo groups the number of silos on the belt and the number of weighing belt conflicts. There are weighing belts conflicts on belts 1, 3, 5 and 6, with most of the conflicts on Belt 3. The solution to the silo grouping ILP provides a silo grouping for which the relaxed model has no weighing belt conflicts, so is in fact integer feasible. Thus for Sloten5, by allowing a change in the silo groupings, the objective value has dropped 0.279% from the best standard model objective value of 1,831,959 to the best relaxed model objective value of 1,826,863 (numbers taken from Section 7.2.3), so the whole previous

loss which was due to weighing belt conflict has been eliminated.

We also formed a hard version of Sloten5, with the addition of 5 extra products. We call this problem Sloten5hard. The standard silo model of Sloten5hard was found to be integer infeasible when solved by ASR1. However, by changing the silo grouping an integer feasible allocation can be found. Indeed, the solution to the silo grouping ILP again gives a silo grouping for which the relaxed model was integer feasible.

When changing the silo groupings it may be beneficial to leave as many silos as possible in their original positions, while still minimising the weighing belt conflicts. For example, of the two equivalent groupings shown in Figure 7.9(b) and Figure 7.9(c) the second one is better, as it moves only two silos from their original positions instead of five. We can model the number of silos moving by adding extra integer variables to the ILP model, that detect if a silo has moved from its original position. The objective is altered to have the twin goals of minimising conflicts and minimising number of silos that move. These two objectives are weighted so that the priority is still minimising conflicts. We call this the *extended silo grouping ILP*. The Sloten5hard problem can be solved with the standard silo grouping ILP and the extended silo grouping ILP. Both solves find a silo grouping with no weighing belt conflicts. The standard ILP solve takes 0.3 seconds by CPLEX (with no branch and bound branches), but moves all 31 silos. The extended ILP solve takes 52.0 seconds by CPLEX (with 1,924 branches), but moves only 12 of the silos.

7.3.2 Unrestricted model

Another new variant of the standard silo model is known as the *unrestricted* model. In this the condition that each silo has a fixed input is relaxed. Each silo can now choose its single input from several alternatives. This leads to further integer constraints. Note that whichever input is chosen for a silo must be used for all of the products. In terms of the silo model given above in (7.1)-(7.11) in the unrestricted model the indicators of silo input, $\boldsymbol{\theta^R}$ and $\boldsymbol{\theta^M}$ are no longer fixed parameters but variables, with a certain θ^R_{rs} or θ^M_{ms} equal to one for the chosen input of silo s. We call the choice of input bins for each silo the *silo allocation*. The unrestricted model has a set of integer constraints for both the silo allocations and the product allocations.

Figure 7.10 shows an example unrestricted problem. All the flow proportion variables have been labelled. $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ describe the production of the mixes. $\boldsymbol{\zeta}$ determines the product allocations, and is bounded by the integer variable $\boldsymbol{\zeta^I}$. $\boldsymbol{\theta^R}$ and $\boldsymbol{\theta^M}$ determine the silo allocations. $\boldsymbol{\kappa}$ are the flow proportion variables for flow direct to the products from the raws. These flows are still possible in the unrestricted model.

The *relaxed unrestricted model* is the unrestricted model with the omission of all the integer constraints. Thus each silo may have multiple inputs, and each product may draw from several silos in each group.

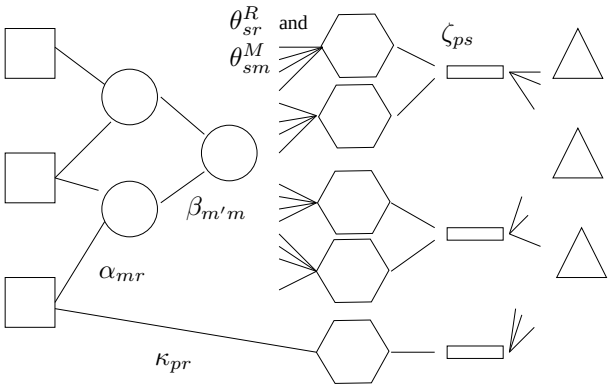


Figure 7.10: Unrestricted version of silo problem Ex1

In the following subsections we attempt to solve the unrestricted model. We first try with the branch and bound MINLP solver MINLP-Fletcher. In order to speed up the solve we create and add *symmetry breaking constraints*. We then consider applying a decomposition method. We then exploit the fact that integer constraints can be converted to (concave) nonlinear constraints to convert the MINLP to an NLP.

Solving with MINLP-Fletcher

We noted above that commercial MINLP solvers were able to (locally) solve the standard silo problem Ex2, but not the Sloten problems. Most MINLP solvers, such as MINLP-Fletcher, use branch and bound, to explore different integer allocations. In silo problems many of the mixes have identical properties, and so are interchangeable. Thus multiple different solutions can occur, which have identical objective value and raw use but appear different as they interchange the use of mixes. This is called *symmetry*. Sherali and Smith [2001] state that symmetry can cause difficulties for branch and bound solvers, as it is inefficient for the solver to enumerate all of the equivalent solutions.

The unrestricted problem is even harder than the standard silo problem, due to the extra integer constraints. As well as symmetry with equivalent mixes there can be symmetry due to equivalent silos, each with the same set of potential inputs and outputs. Furthermore there can be symmetry between the silo groups.

Figure 7.11(a) shows the first half of an example unrestricted silo problem Ex3 with five raws, two mixes, and four silos arranged in two groups of two. In this problem each silo can have any input from Raw 4, Raw 5, Mix 1 or Mix 2. We denote the silo allocation in Figure 7.11(a) as $(m_1r_4)(m_2r_5)$, meaning that Mix 1, Raw 4, Mix 2 and Raw 5 are allocated to Silos 1, 2, 3 and 4 respectively. The silo allocation in Figure 7.11(b) is $(m_1r_4)(r_5m_2)$, which is clearly equivalent, as the two silos in the second group have just been swapped round. The silo allocation in Figure 7.11(c) is $(m_2r_5)(m_1r_4)$, which is also equivalent, as this time we have just swapped the two silo groups around. Changing the order of silo inputs within a group, like

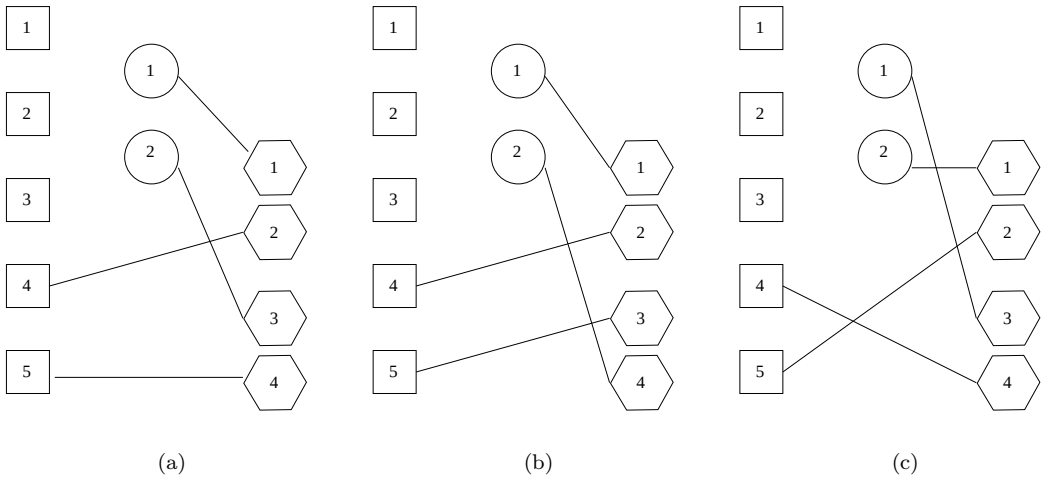


Figure 7.11: First part of unrestricted problem Ex3 with example silo allocation (a) and two equivalent alternatives (b), (c)

in Figure 7.11(b), always gives equivalent allocations. Changing the inputs between two silo groups, like in Figure 7.11(c), gives equivalent allocations as long as the groups are the same size. If two mixes happen to have the same properties, as happens for example in *free flowing* problems where there are no bounds on the mixes, an allocation like $(m_2 r_1)(m_1 r_5)$, which just swaps the mixes, is also equivalent to $(m_1 r_4)(m_2 r_5)$.

In this section we will reduce all these types of symmetry in unrestricted silo models by adding symmetry breaking constraints to the model. There is good evidence for the effectiveness of symmetry breaking constraints. Sherali and Smith [2001] solve an ILP machine assignment problem (for more on assignment problems see Chapter 8), and show that for this problem a reformulation that avoids symmetry solves much more quickly. Margot [2008] shows how symmetry can be detected in ILPs, and suggests several different methods to avoid it; perturbation of the problem, fixing variables, static or dynamic symmetry breaking constraints, and pruning of the branch and bound enumeration tree.

We assume here that all silo groups are of the same size, else a separate rule is needed for each collection of silo groups of a given size. We also assume the problem is free flowing, so all mixing bins are equivalent, else separate rules are needed for each collection of equivalent mixing bins. We also use the fact that there is no cost for using a mix or a silo, so assume that all silos will have some input (even if no product chooses that silo).

In order to keep track of the input to each silo a new variable, $\boldsymbol{I^S}$, is defined below. We call I_s^S the *rank* of silo s.

$$I_s^S = \sum_{r=1}^{R} (M + r)\theta_{sr}^R + \sum_{m=1}^{M} m\theta_{sm}^M \qquad s \in 1 \ldots S. \tag{7.25}$$

These variables are naturally integer valued. Exactly one of the $\boldsymbol{\theta^R}$ and $\boldsymbol{\theta^M}$ variables is nonzero for each silo, as each silo has exactly one input. If silo s has input from a mix then for some m flow proportion variable $\theta^M_{sm} \neq 0$ and $1 \leq I^S_s \leq M$. If silo s has input from a raw then for some r flow proportion variable $\theta^R_{sr} \neq 0$ and $M+1 \leq I^S_s \leq M+R$.

We also define variables $\boldsymbol{I^G}$, for the rank of each group:

$$I^G_g = \sum_{s=1}^{G^{size}_g} I^S_{(G^{start}_g + s - 1)}(M+R)^s \qquad\qquad g \in 1 \ldots G. \tag{7.26}$$

These variables are also naturally integer valued. For each group g the constraint sums up a contribution from each silo in the group. This contribution is the I^S_s of each silo s in the group, each multiplied by the constant formed by $(M+R)$ raised to the power s. These constants are sufficiently large so that each combination of silos in a group gives a different I^G.

Rule	Applies to	
A	within silo group	all mixes placed before all raws (e.g. $(m_1 r_1)$, not $(r_1 m_1)$)
B	within silo group	all mixes numbered in order order (e.g. $(m_1 m_2)$ not $(m_2 m_1)$
C	within silo group	all raws are placed in lexicographic order (e.g. $(r_1 r_2)$ not $(r_2 r_1)$
D	between silo groups	silo groups are ordered by the silos they contain
E	mixes	mixes ordered by weight (e.g. m_1 the lightest and so on)

Table 7.8: Rules for stopping equivalent solutions

Table 7.8 details five rules that limit the number of silo allocations to a minimal, canonical set. Rules A-C govern the ordering of mixes and raws within each silo group. Rule D governs the ordering between groups. Rule E applies to the mixes, and prevents symmetries that can occur from equivalent mixes. We consider the three sets of rules (A-C, D and E) in turn.

We can force the solution to satisfy Rules A-C with the linear constraint below, which orders the silos by rank from low to high within each group. Note that this inequality does not apply to silos at the start of each group, so only enforces an ordering of silos within each group, and not between groups. We use a strict inequality to forbid two silos in the same group having the same rank, as having two silos in a group with the same input is never optimal.

$$I^S_{s-1} < I^S_s \qquad\qquad g \in 1 \ldots G,\ G^{start}_g + 1 \leq s \leq G^{start}_g + G^{size}_g. \tag{7.27}$$

We can force the solution to satisfy Rule D with the linear constraint below, which orders the group by their ranks. We use a non strict inequality here as it may sometimes be optimal

to have several identical silo groups.

$$I^G_{g-1} \leq I^G_g \qquad g \in 2 \ldots G. \tag{7.28}$$

Finally to account for mixing bin equivalence and so satisfy Rule E we add the linear constraint below. Recall that $\boldsymbol{u}$ is the mass of the mixes. This constraint orders all the mixes by weight, forcing the lowest numbered mix to be the lightest, and so on. Rule E could alternatively be enforced by comparing characteristics of the mixes other than the weight, for example nutrient concentration of a given nutrient.

$$u_{m-1} \leq u_m \qquad m \in 2 \ldots M. \tag{7.29}$$

We solve the unrestricted silo problem Ex3 with the addition of constraints (7.25)-(7.26) to define the rank of each silo and group, and constraints (7.27)-(7.29) to limit symmetry in silo input, group input and mix equivalence. With the addition of the symmetry breaking constraints there are only nine canonical solutions to Ex3, $(m_1m_2)(m_1r_4)$, $(m_1m_2)(m_1r_5)$, $(m_1m_2)(m_2r_4)$, $(m_1m_2)(m_2r_5)$, $(m_1m_2)(r_4r_5)$, $(m_1r_4)(m_2r_4)$, $(m_1r_4)(m_2r_5)$, $(m_1r_5)(m_2r_4)$ and $(m_1r_5)(m_2r_5)$. Note that only one of the three equivalent solutions shown in Figure 7.11, $(m_1r_4)(m_2r_5)$, is included in this list.

	variables	constraints	B&B branches	time
Standard solve	101	67	44.0	0.36
With symmetry breaking constraints	107	77	14.4	0.23

The table above shows the results for solving Ex3 with and without the addition of the symmetry breaking constraints that we developed above, using the branch and bound MINLP solver MINLP-Fletcher. The first two columns give the number of variables and constraints. There are slightly more variables and constraints with the addition of symmetry breaking constraints. The final two columns give details of the solve, averaged over five runs. With the addition of the symmetry breaking constraints there were on average fewer branch and bound branches and a shorter average solve time. This methods has not been tried on larger problems, such as the Sloten factories. For these problems it is anticipated that, due to the expected saving in number of branches, solves with symmetry breaking constraints will be a lot faster than those without. However larger unrestricted problems may be too hard to solve at all with branch and bound, even with symmetry breaking constraints.

Decomposition

We may attempt to solve the unrestricted model by solving a simpler series of MINLPs, in which either the silo allocations or the product allocations are fixed. Fixing the silo allocations gives a standard silo model, which could be solved by ASR1, to give new product allocations. Fixing the product allocations gives a MINLP similar to the standard silo model, which could be solved by a modified version of ASR1, to give new silo allocations. The resulting algorithm of alternately solving these two types of simpler MINLP is a decomposition, where each subproblem is of the same difficulty as a standard silo model.

Replacing integer constraints with concave costs

We now attempt to solve unrestricted problems using a new heuristic algorithm that we call AUR1. This algorithm is motivated by the potential difficulty of using branch and bound solvers on the unrestricted model, due to the very large number of integer variables in these problems.

Algorithm AUR1 works by solving the relaxed unrestricted model, which is an NLP with no integer constraints. We then solve multiple NLPs each with the addition of concave costs in the (minimisation) objective that attempt to force integrality in the binary variables (and in our formulation of the unrestricted model all integer variables are binary variables). This method is an attempt to find a good local minimum, rather than to efficiently enumerate all solutions, so does not need the symmetry breaking constraints developed above. An outline of the algorithm is given in Table 7.9.

1	Solve relaxed unrestricted NLP.
2	Solve a series of NLPs with concave costs instead of integer constraints.
3	Fix silo allocations.
4	Solve a series of NLPs with concave costs instead of integer constraints (for product allocations only).
5	Fix product allocations.

Table 7.9: Algorithm AUR1 for unrestricted silo problem

In Step 1 we solve the relaxed unrestricted model. This is an NLP, and in fact all the solves of this algorithm are just NLP solves. At the solution to the relaxed unrestricted model it is likely that there are several fractional binary variables, so the problem is integer infeasible.

In Step 2 we therefore add concave costs to the NLP objective that push the fractional binary variables towards integrality, by rewarding these variables for their distance from 0.5. Let $\boldsymbol{x}$ represent a general vector of all the variables of the problem, and $\boldsymbol{\epsilon}$ be a scalar parameter that determines the relative magnitude of the concave costs for different fractional binary variables. We refer to it as the *curvature*. The concave cost added to the objective for each fractional binary variable is:

$$-\epsilon_i(x_i - 0.5)^2. \tag{7.30}$$

$\boldsymbol{\epsilon}$ is a scalar parameter that determines the relative magnitude of the concave costs for different fractional binary variables. The potential danger of using concave costs is that when the curvature is large it does not allow different integer allocations to be explored. The curvature must therefore be correctly weighted so that the NLP is only gently guided by the concave costs. After one solve with the addition of concave costs if the resulting solution is still not integer feasible the curvature is increased on any fractional binary variables that are still fractional, and curvature is added to any new fractional binary variables. When a fractional binary variable becomes integer we do not remove the the concave cost for that variable, to prevent cycling. There may be several NLP solves in Step 2, with the curvature gradually increasing after each one.

If the curvature across all variables gets sufficiently high without forcing an integer feasible solution Step 3 is entered. We reset $\boldsymbol{\epsilon}$ to zero and instead fix every binary variable in the first part of the problem to be either 0 or 1. Recall that the fractional binary variables in the first part correspond to silos with multiple inputs. To fix these variables to either 0 or 1 the largest input proportion to each silo (which may be less than 0.5) is fixed at 1, and the other inputs to that silo are fixed at 0. After fixing all the integer variables the NLP may be infeasible, in which case the algorithm has failed, and must be restarted with a different random start point.

If the NLP is still feasible we enter Step 4, and repeatedly resolve an NLP with curvature added to fractional binary variables in the second part. If after repeatedly resolving the NLP with increased curvature no integer feasible solution is found Step 5 is entered. $\boldsymbol{\epsilon}$ is again set to zero and all integer variables in the second part of the problem are also fixed, using a similar method to the one used in the first part. This may also make the NLP infeasible, in which case the algorithm has failed.

Figure 7.12(a) shows the solution to the relaxed model of Ex3, at the end of Step 1. These are labelled with the actual flow proportions. There are integer violations in Silo 4, which is fed from a mix and a raw, and Product 1, which uses both silos from the first silo group. In Step 2 AUR1 adds concave costs to the objective for these two fractional binary variables and resolves the NLP. This gives a new solution that is also integer infeasible. The curvature is repeatedly increased until after three iterations we get an integer feasible solution, shown in Figure 7.12(b). This is in fact the global solution for Ex3. AUR1 is therefore successful on Ex3, finding the optimal solution in Step 2 of the algorithm, before having to fix any integer variables in either part of the problem. AUR1 has not been tested on larger problems.

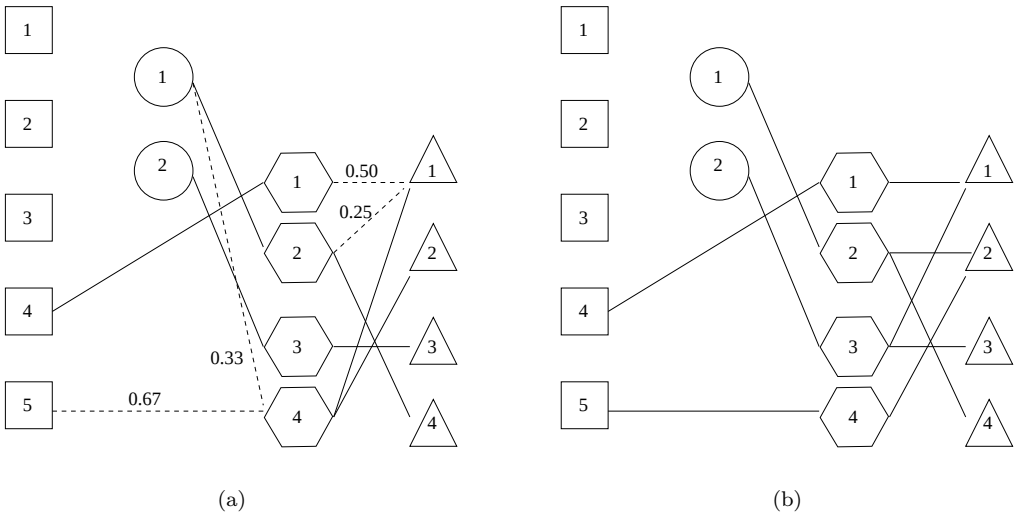


Figure 7.12: Ex3 relaxed model solution (a) and final solution (b)

7.4 Summary

In this chapter we have introduced the silo problem, an MINLP based on the Sloten factory. Algorithm ASR1 was developed to solve the standard silo model. It decomposes the MINLP into an NLP and ILP. The ILP can be further decomposed by the dual cutting plane method, though this is problematic due to the duality gap. Solving the Sloten problems using ASR1 was found to take only two or three times longer than solving without the integer constraints. Furthermore the objective values were also only slightly affected by the integer constraints.

The silo grouping problem allows the grouping of the silos to be altered. This was approximately solved for the Sloten problems, and was able to make the relaxed standard silo model integer feasible.

The unrestricted model is a generalisation of the standard model, that allows each silo to choose one of several possible inputs. We introduced symmetry breaking constraints which enabled an example unrestricted problem to be solved more quickly by branch and bound. An algorithm AUR1 was developed to solve unrestricted problems by adding concave costs to fractional binary variables, which was able to solve a small example problem.

7.4.1 Further work

For the standard silo model ASR1 has only been tested on the Sloten problems. If they become available it would be beneficial to test ASR1 on other problems, or even other versions of the Sloten factory. There is also the potential for much more work on the silo grouping and unrestricted model, if Sloten International decide that these are the problems that they wish to solve.

Chapter 8

Cattle Mating Problem

The cattle mating problem (CMP) first came to our attention with the MSc thesis of Horgan [2007]. The thesis reports work with the Irish Cattle Breeding Federation (ICBF) on determining the values of some parameters that appear in the problem. The ICBF then requested further work on solving the CMP.

In Section 8.1 we introduce assignment problems, of which the CMP is a special case. Briefly, the CMP is the problem faced by a farmer of choosing which cows in the herd to mate with which bulls (though actually all breeding is done by artificial insemination). Different versions of the CMP arise depending on the specific objectives of the farmer, and how realistically the problem is modelled. In Section 8.2 we describe the various versions of the CMP that can occur, including what we call the *standard CMP model*, which is an ILP. This is the problem that the ICBF wish to solve. In Section 8.3 we develop a new solver named Cowculus, that heuristically solves the standard CMP model by decomposing it into a series of LPs.

8.1 Assignment problems

We now introduce assignment problems. It is convenient throughout this chapter to use the language of the CMP to describe all problems, rather than the original terminology of each problem. We therefore always think of cows to bulls, rather than jobs and machines or anything else. We refer to *assigning* a cow to a bull (and not the other way round), to produce a particular calf. Typically all the cows are used exactly once in the solution, and bulls may be used any number of times (including not at all). We use the term *selected* to refer to any bull that is *used* at least once in the solution.

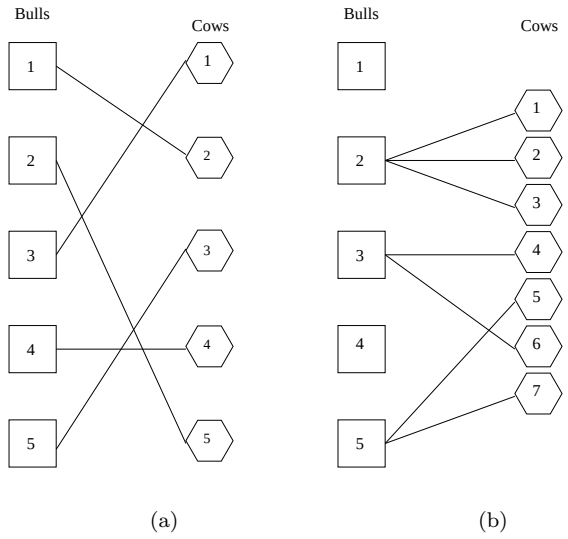


Figure 8.1: Example solutions to basic assignment problem (a) and generalised assignment problem (b)

$$\max_{\boldsymbol{x}} \sum_{b=1}^{B} \sum_{c=1}^{C} S_{bc} x_{bc} \tag{8.1}$$

$$\text{s.t.} \sum_{b=1}^{B} x_{bc} = 1 \qquad c \in 1 \dots C, \tag{8.2}$$

$$\sum_{c=1}^{C} x_{bc} = 1 \qquad b \in 1 \dots B, \tag{8.3}$$

$$0 \leq x_{bc} \qquad b \in 1 \dots B,\ c \in 1 \dots C. \tag{8.4}$$

The simplest assignment problem is the *basic assignment problem* (BAP), for which the mathematical formulation is given above. Bulls are indexed by $b \in 1 \dots B$ and cows by $c \in 1 \dots C$. The variable x_{bc} records if bull b mates with cow c, and S_{bc} is the score for the calf resulting from such a mating (or, equivalently, we can consider it the score for the mating itself).

The objective, (8.1), is to maximize total profit, which is the sum of the scores of all calves that are produced. Constraint (8.2) is here referred to as the *cow use constraint* and enforces the condition that each cow is assigned to exactly one bull. Constraint (8.3) is here referred to as the *bull use constraint*, and enforces the condition that each bull must be assigned exactly one cow. These two constraints seem to imply that BAPs can only be solved when there is the same number of bulls and cows. However, with reformulation by the introduction of dummy bulls or cows, problems with unequal numbers of cows and bulls can also be modelled as BAPs. If there are too many bulls (say) then dummy cows are created and assigned to the spare bulls with zero scores. Constraint (8.4) ensures that $\boldsymbol{x}$ is non negative. With the equalities (8.2) and

(8.3) changed to inequality form (with $\leq$ instead of $=$) the model is fundamentally unchanged, and we still refer to it as the BAP.

Figure 8.1(a) shows an example solution to a BAP. Cow 1 is assigned to Bull 3, Cow 2 is assigned to Bull 1 and so on. It is a feature of assignment problems that often there are a great number of feasible solutions, and finding an integer feasible solution can be relatively easy. This is in contrast to the silo problem of Chapter 7, where feasible solutions are hard to find. In the solution to a BAP $\boldsymbol{x}$ must be integer valued, though there is no constraint directly enforcing this. However, the model has a constraint matrix that has the property of being *unimodular*, and has integer right hand sides to the constraints. These two properties mean that the solution to the linear relaxation of the BAP model (8.1)-(8.4) has integer solutions (see e.g. Winston [2000] page 525). BAPs can therefore be solved as LPs by the simplex method, without explicitly enforcing integrality of $\boldsymbol{x}$, and the resulting solution is optimal. Other solve methods include the transportation simplex method and the Hungarian method.

The *generalised assignment problem* (GAP) was introduced by Ross and Soland [1975]. It is a generalisation of the BAP, or, equivalently, the BAP is a special case of the GAP. The mathematical formulation of the GAP is given below.

$$\max_{\boldsymbol{x}} \sum_{b=1}^{B} \sum_{c=1}^{C} S_{bc} x_{bc} \tag{8.5}$$

$$\text{s.t.} \sum_{b=1}^{B} x_{bc} \leq 1 \qquad c \in 1 \ldots C, \tag{8.6}$$

$$\sum_{c=1}^{C} A_{bc} x_{bc} \leq U_b \qquad b \in 1 \ldots B, \tag{8.7}$$

$$x_{bc} \in \{0, 1\} \qquad b \in 1 \ldots B,\ c \in 1 \ldots C. \tag{8.8}$$

A_{bc} is a coefficient for the weighting of bull b with cow c (in the cattle mating problem it is usual that $A_{bc} = 1$, but this may not be the case when other types of matching are considered). U_b is the upper bound on uses of bull b (typically integer in the cattle mating problem, but may be noninteger when other types of matching are considered). Most of the model is the same as the BAP. The objective (8.5) is unchanged. The cow use constraint (8.6) is now an inequality as not all cows must be used. The bull use constraint, (8.7), has been generalised to say that the sum of cows assigned to bulls, each weighted with a real coefficient, must be less than some other real number. This means that potentially several cows can be assigned to one bull. Figure 8.1(b) shows an example solution to a GAP, which demonstrates this.

In the model of a GAP the constraint matrix is no longer unimodular, hence the solution of the linear relaxation is typically fractional. Integrality of $\boldsymbol{x}$ must therefore be explicitly enforced by the model, which is done by (8.8). A GAP is therefore an ILP. Section 7.1.2 contained a brief

discussion of ILP solvers, in particular branch and bound. There are also many ILP methods specialised for GAPs. These are summarised in e.g. Osman [1975] or Wilson [2005]. We now describe some of them which are relevant to Cowculus, using the language of the CMP. They are all heuristic methods, which typically solve the problem approximately. Like the MINLP solve methods given in Section 7.1.2, the first solution found is often then refined local improvement methods.

Foulds and Wilson [1997] solve GAPs with a *regret minimisation algorithm* and also with a *greedy algorithm*. In the regret minimisation algorithm they first order the cows by the difference between the mating scores of the best bull for that cow and the second best bull. Starting with the cow with the largest difference in mating values each cow is assigned the best available bull, which approximately solves the problem. The solution is then improved by using what are called *2-opts*. In the language of the CMP a 2-opt swaps the bull assignments of two cows. For example, looking at Figure 8.1(a) after a 2-opt Cows 1 and 2 might be reassigned to Bulls 1 and 3 respectively. he greedy algorithm begins by solving a relaxed version of the GAP without the bull use constraint (8.7). This usually gives a solution that has excellent objective value but is infeasible by assigning too many cows to some bulls. This solution is then improved, in feasibility, by using *1-opts*. A 1-opt assigns a cow to a different bull. For example, looking at Figure 8.1(a) after a 1-opt Cow 1 might be reassigned to Bull 1. The greedy algorithm was found to be more effective than the regret minimisation algorithm.

Amini and Racer [1995] assign cows to bulls using a combination of a regret minimisation algorithm and a greedy algorithm, then locally improve the solution using what they calls a *variable depth first heuristic*.

Cohen et al. [2006] consider the bulls one at a time. They first assign some number of cows to the first bull, by solving a small ILP known as a *knapsack problem*. Based on the cows assigned to the first bull the objective costs on matings involving the unassigned cows are adjusted. Using these new objective costs some number of cows are assigned to the second bull, again by solving a knapsack problem, and so on through all the bulls.

Trick [1992] uses the linear relaxation of the GAP model, which omits the integer constraint (8.8). In each iteration of the algorithm the linear relaxation is solved by LP, then based on the solution some integer variables are eliminated from the problem and some are fixed. In successive iterations more integer variables are eliminated or fixed until an integer feasible solution is found, which is then improved by several local improvement methods; 1-opts, 2-opts and even resolving small parts of the problem. Trick shows that the linear relaxation of a GAP has a limited number of fractional integer variables in the solution, and so is not far from being integer feasible.

French and Wilson [2007] solve a similar problem to the GAP, also with an algorithm based on linear relaxation. As in Trick [1992] in each iteration an LP is solved and in successive iterations more integer variables are fixed, until an integer feasible solution is found, which is

then improved by local improvement methods.

The Cowculus solver is also based on linear relaxation, and also repeatedly solve linear relaxations and fixes the value of integer variables as the solve progresses, but using a different method for fixing integer variables to Trick [1992] and French and Wilson [2007]. Like Trick for the GAP we found that for the standard CMP model, that we introduce later, the linear relaxation solution was close to being integer feasible, and had many of the same matings as in the optimal integer feasible solution. The work of Foulds, where a greedy algorithm is more effective than a regret minimisation algorithm, is also encouraging for our Cowculus solver, which solves in a similar way to the greedy algorithm.

8.2 Cattle mating problem

The cattle mating problem has many different versions. Section 8.2.1 describes three types of model. The most complex of these, Model 3, is referred to as the *standard CMP model*, and is the one solved by Cowculus. Section 8.2.2 describes the standard CMP model in detail. Section 8.2.3 introduces some example problems that we will later solve.

8.2.1 Types of model

Model 1

This is the most basic model. Each cow is assigned to at most one bull, and there are no other constraints. This is therefore a simplification even of the BAP (and therefore also of the GAP), with the same mathematical formulation as the BAP but with the inequality form of (8.2) and without constraint (8.3). The model decomposes by cow, and can be solved by simply assigning each cow to the best bull for that cow.

Model 2

This model is as Model 1 but includes a constraint limiting the number of uses of each bull, with an integer valued upper bound U_b on uses of bull b. The resulting mathematical model is a GAP, but with a simple form of the bull use constraint (8.7), where $A_{bc} = 1$ for all b and c. Problems of this type can be reformulated by making U_b copies of each bull b. With this reformulation each new bull is used at most only once. This means that the bull use constraint can be written in the inequality form of (8.3), and the problem represented as a BAP. Since the problem can be reformulated as a BAP it can be solved to integer optimality by the simplex method (without actually having to do the reformulation), or any of the other methods used to solve BAPs.

Model 3

This model includes lower and upper bounds, L_b and U_b, on the number of times each selected bull b can be used (recall a selected bull is one that is used at least once). The lower bound is a consequence of the bull semen being delivered in packets of size greater than one (typically five), and the upper bound is included to minimise the risk to the farmer that one of the selected bulls will produce unexpectedly poor offspring.

There are also lower and upper bounds, $\underline{\delta}$ and $\overline{\delta}$, on the total number of bulls selected. The lower bound $\underline{\delta}$ is also to minimise the risk to the farmer from over-reliance on any one bull. It is suggested in Nash and Rogers [1996] that the common recommendation is to select between three and seven bulls, irrespective of the number of cows in the herd, and indeed that is also their finding. The upper bound $\underline{\delta}$ on number of bulls selected is to prevent the cost associated with selecting many different bulls.

Because of the two extra sets of bounds Model 3 is neither a BAP or a GAP. It has a third form, which we describe below. This new model is the one that is solved by Cowculus, and it is henceforth referred to as the standard CMP model.

8.2.2 Standard CMP model

B	number of bulls, indexed with b
C	number of cows, indexed with c
L	minimum number of bulls to be selected
U	maximum number of bulls to be selected
L_b, U_b	minimum and maximum number of times bull b can be used, if selected
$\underline{\delta}$,$\overline{\delta}$	lower and upper bounds on number of bulls selected
S_{bc}	score of calf from mating of bull b with cow c
W_{bc}	binary parameter permitting bull b mating with cow c
x_{bc}	binary variable for bull b mating with cow c
δ_b	binary variable for selecting bull b

Table 8.1: Structural and other parameters (all upper case) and variables (lower case) used in standard CMP model

In this section we define in full the standard CMP model, introduced as Model 3 above. Table 8.1 shows the parameters and variables used in the solve.

$$\max_{\boldsymbol{x},\boldsymbol{\delta}} \sum_{b=1}^{B}\sum_{c=1}^{C} S_{bc}x_{bc} \tag{8.9}$$

$$\text{s.t.} \sum_{b=1}^{B} x_{bc} \leq 1 \qquad c \in 1 \ldots C, \tag{8.10}$$

$$\delta_b L_b \leq \sum_{c=1}^{C} x_{bc} \leq \delta_b U_b \qquad b \in 1 \ldots B, \tag{8.11}$$

$$\underline{\delta} \leq \sum_{b=1}^{B} \delta_b \leq \overline{\delta}. \tag{8.12}$$

The model is defined in (8.9)- (8.12). The objective, (8.9), is the same as for the BAP and the GAP.

The cow use constraint, (8.10), still says that each cow is assigned to a maximum of one bull. The bull use constraint, (8.11), says that if bull b is selected ($\delta_b = 1$) it must be used at least L_b and at most U_b times, and if it is not selected ($\delta_b = 0$) it must be used zero times. This constraint is unlike the bull use constraint of BAP and GAP, as it has lower and upper bounds on bull use controlled by a new integer variable. Constraint (8.12) is known as the *bull selection constraint*, and says that the number of bulls selected must be at least $\underline{\delta}$ and at most $\overline{\delta}$. As well as these constraints we have simple bounds on the variables, in particular each mating x_{bc} is constrained to occur a maximum of W_{bc} times.

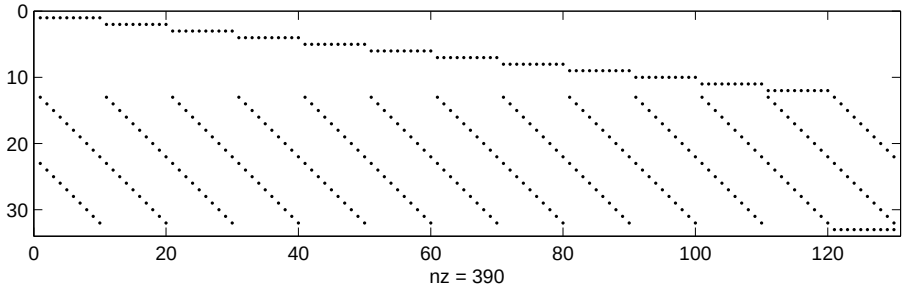


Figure 8.2: Sparsity pattern of constraint matrix for Mininew for standard CMP model (a)

Figure 8.2 shows the sparsity pattern of the constraint matrix of the standard CMP problem Mininew, which has $B = 10$ bulls and $C = 12$ cows. The constraint matrix has $C+2B+1 = 33$ rows; $C = 12$ for the cow use constraints of (8.10), then $2B = 20$ for the bull use constraints of (8.11), then one for constraint (8.12). There are $BC + B = 130$ columns; $BC = 120$ for $\boldsymbol{x}$, then $B = 10$ for $\boldsymbol{\delta}$. There are $3BC + 3B = 390$ nonzeroes, all of which have values of 1, apart from the nonzeros in the final 10 columns excluding the bottom row, which have values corresponding to $\boldsymbol{L}$ and $\boldsymbol{U}$.

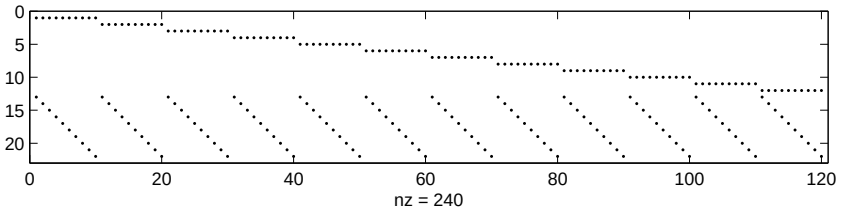


Figure 8.3: Sparsity pattern of constraint matrix for Mininew for δ-relaxed CMP model

$\boldsymbol{\delta}$-relaxed CMP model

The standard CMP model is an ILP, with two sets of integer variables, $\boldsymbol{x}$ and $\boldsymbol{\delta}$. We now consider relaxing the model by dropping one set of integer variables. We call the standard CMP model with $\boldsymbol{\delta}$ variables dropped the *$\boldsymbol{\delta}$-relaxed CMP model*. In the $\boldsymbol{\delta}$-relaxed CMP model constraint (8.12) is removed altogether, and bull use constraint (8.11) is relaxed to become:

$$0 \leq \sum_{c=1}^{C} x_{bc} \leq U_b \qquad\qquad b \in 1 \ldots B. \tag{8.13}$$

In this new version of the constraint for all bulls the lower bound, $\delta_b L_b$, has been replaced with its minimum value of 0, and the upper bound, $\delta_b U_b$, has been replaced with its maximum value of U_b. The new constraint is linear. Figure 8.3 shows the sparsity pattern of the constraint matrix to the $\boldsymbol{\delta}$-relaxed CMP model of MiniNew, which is slightly smaller than the constraint matrix of the original problem.

The $\boldsymbol{\delta}$-relaxed CMP model is still an ILP due to the binary $\boldsymbol{x}$ variables. It is in fact a simple instance of GAP, and were it not for (8.13), which has upper bounds which can be greater than one, it could immediately be identified as an instance of BAP. It is indeed an instance of BAP, as can be seen by reformulating the model in the same way that we reformulated Model 2 above by duplicating bulls (though this reformulation is not necessary in a solve). As the $\boldsymbol{\delta}$-relaxed CMP model is equivalent to the BAP we do not need to enforce integrality of $\boldsymbol{x}$, and it can be solved as an LP. This is of vital importance to Cowculus, as the solver works by repeatedly solving the $\boldsymbol{\delta}$-relaxed CMP model with a simplex solver.

Advanced CMP model

A minor variant to the standard CMP model is known as the *advanced CMP model*. In this the farmer specifies in advance some (or all) of the bulls that are to be selected. This variant does not fundamentally change the model. If the farmer pre-selects all bulls then all that remains to be done is assign each cow to a bull, and the model simplifies to being of type Model 2, which can be solved by LP.

Calf scores

S_{bc}	Calf score of calf from bull b and cow c
E_b^B,E_c^C	European Breeding Index (EBI) of bull b and cow c respectively
M_b	semen price of semen from bull b
I_{bc}	inbreeding coefficient between bull b and cow c
H_{bc}	heterosis coefficient between bull b and cow c
P_{bc}	expected difference between production and fertility of calf from bull b and cow c

Table 8.2: Parameters for determining calf scores

We now look in more detail at how the calf scores are generated. These scores are estimates for the value of each calf. The scores are parameters in the model, and for the Cowculus solver are assumed to have already been generated. However, how the scores are generated is important, as it affects the difficulty of each of the problems solved.

In Europe the value of an animal is traditionally equated to the European Breeding Index (EBI) of that animal. The EBI of an animal is made up of several weighted subindices. These are; production, fertility, calving ease, beef and health. Of the subindices production and fertility have the largest weighting. The EBI of a calf is the average of the EBI of its parents. One way to calculate the score of a calf is to use the formula below:

$$S_{bc} = \frac{E_b^B + E_c^C}{2} - M_b \qquad b \in 1 \dots B,\ c \in 1 \dots C. \tag{8.14}$$

where the score for a calf is the average of the EBIs of its parents minus the semen price. Using these calf scores makes a problem trivial to solve, as the scores are separable by bull. In the optimal solution the best bull is used the maximum number of times, then the second best bull is used the maximum number of times, and so on, and any assignment of cows that uses these best bulls is optimal.

In practice more complicated formulas for generating the calf scores are used. Nash and Rogers [1996] in the USA estimate bull quality using a combination of semen price, expected milk levels of fat milk and protein in daughters, and standardised transmitting abilities of foot angle, udder depth and teat placement. Together these give the expected Net Revenue of a bull. In Horgan [2007] the value of a calf is estimated as:

$$S_{bc} = \frac{E_b + E_c}{2} - M_b - 2.5I_{bc} + 0.24H_{bc} - 0.5P_{bc}, \qquad b \in 1 \dots B,\ c \in 1 \dots C, \tag{8.15}$$

where $\boldsymbol{I}$, $\boldsymbol{H}$ and $\boldsymbol{P}$ are further parameters for expected calf quality. $\boldsymbol{I}$ is the inbreeding, which is a decrease in the robustness of a calf due to genetic similarity of the parents. The deleterious effect of inbreeding increases nonlinearly as the level increases. If the inbreeding level of a calf

is above a certain threshold (10% is suggested by Horgan) then it may be best to disallow production of that calf in the model. In McParland et al. [2007a] and McParland et al. [2007b] inbreeding in the Irish cattle population is shown to be at low levels but increasing. $\boldsymbol{H}$ is the level of heterosis, also known as hybrid vigour, which is an increase in the robustness of a calf from having unrelated parents. $\boldsymbol{P}$ is the expected difference between production and fertility of calves. It is weighted negatively in (8.15), as a high difference in expected production and fertility makes a calf score less reliable, and farmers are risk averse.

Due to the difficulty of generating inbreeding and heterosis levels it is known that the ICBF sometimes use a simplified version of (8.15):

$$S_{bc} = \frac{E_b + E_c}{2} - 2M_b - 0.5P_{bc} \qquad b \in 1 \dots B,\ c \in 1 \dots C. \tag{8.16}$$

In summary, it is believed that the calf scores that are normally used by the ICBF, including in the test problems below, are dominated by the EBI component. However these calf scores are not separable by bull, and so the problems do require a full ILP solve.

8.2.3 CMP test problems

	B	C	variables	constraints
Mininew	10	12	130	33
Advrunex	3	157	474	164
Prerunex	110	214	23,540	324
Prerunex2	110	214	23,540	324

Table 8.3: Dimensions of test problems

Table 8.3 gives the number of bulls and cows in the four standard CMP model test problems, and the size of the ILP models, using the standard CMP model given above. For each problem the minimum and maximum number of bulls that can be selected is equal, at 5, 3, 8 and 8 bulls respectively.

Mininew is a small test problem invented for testing Cowculus. Advrunex is a real ICBF problem which has only three bulls. These three bulls have been pre-selected by the farmer, so this is an example of an advanced CMP model. It can therefore be solved by LP, though Cowculus (and other solvers) still solve it as an ILP.

Prerunex is also a real ICBF problem, that has a large number of cows and bulls. Prerunex2 is identical to Prerunex apart from the calf scores. It is believed that the calf scores of ICBF problems like Prerunex are dominated by the EBI component, so the best bulls will be best for nearly all the cows. This means picking which bulls to select is fairly easy. In Prerunex2 the calf score for bull b and cow c is generated by the quasi-random function $S_{bc} = 2 + \sin(b) + \cos(c)$, so different bulls are better for different cows. This is done to make Prerunex2 a more challenging

test problem.

The frequency and number of optimisations performed and costs involved to the farmers, and the corresponding savings from an optimised solution, are not known to us.

Finding reserve bulls

As well as finding the optimal assignment of cows to bulls, the farmer may also require good 2nd and 3rd choice bulls for each cow, so if for some reason the 1st choice bull is unavailable a reserve bull can be used. To solve the problem with the extension of finding reserve bulls we could introduce extra integer variables to the model to determine if a calf is a 1st 2nd or 3rd choice, and alter the objective function to include some weighting between the importance of 1st, 2nd and 3rd choice calves. However this would make the problem much harder to solve, so is not done in Cowculus.

8.3 Cowculus solver

The motivation for the solver Cowculus is that the ICBF wish to be able to solve standard CMP problems reasonably quickly, in no more than about ten seconds per problem, and reasonably accurately. A very fine degree of accuracy is not required, as the calf scores in the data are only estimates of quality and are not precise.

As the standard CMP model is an ILP it can be solved very quickly and accurately by a commercial solver such as CPLEX. However, there are three advantages to instead using a heuristic method such as Cowculus. Firstly, it avoids the cost of a commercial solver. Secondly, by using a heuristic that gradually approaches the solution, we can guarantee to find a (approximate) solution within a given time frame. Although it is usually fast there is a possibility that CPLEX will still be deep in calculation after ten seconds. Thirdly, with a heuristic like Cowculus, we can easily adapt the algorithm to cater for further developments in the problem, for example the problem of finding reserve bulls.

In Section 8.3.1 we describe the Cowculus algorithm. In Section 8.3.2 we give results for solving the test problems with Cowculus, including comparing it with other ILP solvers.

8.3.1 Algorithm

Cowculus is a heuristic solver for the standard CMP model. In the main algorithm it solves the standard CMP model. It then addresses the problem of finding reserve bulls.

Before starting the solve Cowculus analyses the problem data, and detects any anomalies. Mild anomalies, such as a bull having a minimum number of uses of a negative number, are merely flagged up. Serious anomalies, such as a bull having a lower bound on number of uses greater than the upper bound, cause the code to abort with an error message.

Main algorithm

1 Solve the $\boldsymbol{\delta}$-relaxed CMP model.
2 Solve a series of altered $\boldsymbol{\delta}$-relaxed CMP models to progressively lower the number of bulls selected (call this Step 2a) or raise it (call this Step 2b), until an acceptable number is selected.
3 Solve a final altered $\boldsymbol{\delta}$-relaxed CMP model that ensures each of the selected bulls is used an acceptable number of times.

Table 8.4: Cowculus algorithm to solve standard CMP model

The standard CMP model has two sets of binary variables, $\boldsymbol{x}$ for the cow-bull allocations, and $\boldsymbol{\delta}$ for whether or not each bull is selected. The Cowculus algorithm works by relaxing the $\boldsymbol{\delta}$ variables, to form the $\boldsymbol{\delta}$-relaxed CMP model. While still an ILP, this model has the favourable property that it can be solved by LP. However it does not include constraint (8.12), which limits the total number of bulls selected, and uses a relaxed version of constraint (8.11), which ensures that each selected bull is used an acceptable number of times. These two constraints are not properly enforced in the $\boldsymbol{\delta}$-relaxed CMP model, so must be enforced heuristically by Cowculus, which is done between the $\boldsymbol{\delta}$-relaxed CMP model solves. All these solves are LPs, for which we use the LP solver EMSOL. We refer to forming and solving each $\boldsymbol{\delta}$-relaxed CMP model as an *iteration*. All iterations apart from the first are warm started.

An outline of the algorithm is given in Table 8.4. There is just one iteration in Step 1, potentially many in Step 2, and one in Step 3. In Step 1 we solve the original $\boldsymbol{\delta}$-relaxed CMP model. In Step 2 we progressively enforce (8.12), by fixing some of the $\boldsymbol{\delta}$ variable. By the end of Step 2 $\boldsymbol{\delta}$ is entirely fixed, and (8.12) is satisfied. With $\boldsymbol{\delta}$ fixed (8.11) is now linear. Thus in Step 3 we restore (8.11) and solve once more, so the lower and upper bounds for all the selected bulls are enforced (this step is not necessary if these bounds are already satisfied). Thus the final solution at the end of Step 3 satisfies all the constraints of the standard CMP model. If after any iteration in Step 2 the suggested time limit (typically 10 seconds) has been exceeded then we immediately force the number of bulls selected to either $\underline{\delta}$ or $\overline{\delta}$, and the algorithm moves to Step 3.

		use of each bull										objective value
		1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	
Iteration 1	Step 1	-	-	-	-	1	1	3	2	1	2	122.0
Iteration 2	Step 2a	-	-	-	-	1	-	3	3	1	2	115.0
Iteration 3	Step 2a	-	-	-	-	-	-	3	4	1	2	109.0
Iteration 4	Step 3	-	-	-	-	-	-	2	4	2	2	108.0

The example solve above shows how the use of each bull might change each iteration. In this problem a maximum of four bulls can be selected, and each selected bull must be used at least twice. Step 1 consists of one iteration, at the end of which too many bulls are selected, and some of those that are selected are not used enough. Thus both (8.12) and (8.11) are violated.

Step 2 consists of two iterations, at then end of which an acceptable number of bulls is selected, so (8.12) is satisfied, but still Bull 9 is used too few times. Thus in Step 3 we perform one iteration so that (8.11) is also satisfied. We then have a feasible solution, which hopefully also has good objective value.

The crux of the algorithm is the method for lowering or raising the number of bulls selected, in either Step 2a or Step 2b. In both cases we change the number of bulls selected by only one each iteration, in the hope that the gradual move to an acceptable number of bulls gives a better chance of selecting the best bulls.

In Step 2a we lower the number of bulls selected, by *deselecting* a bull b that is used the least. It is possible that when choosing which bull to deselect two bulls are used the same number of times. In this case we tie break on the contribution to the total profit from each bull, deselecting the bull that gives the least total profit. Making the contribution to the total profit from each bull the primary indicator of which bull to deselect was also tried, but was found to be a worse indicator than the number of bull uses. To deselect bull b we set the upper bound on use, U_b in (8.13), to zero. This can be thought of as setting $\delta_b = 0$ in the original constraint (8.11). All the bulls that were not selected in the last iteration also have their upper bound on use set to zero, to guarantee that the solution to the next $\boldsymbol{\delta}$-relaxed CMP model selects (at least) one fewer bull.

In Step 2b we raise the number of bulls selected, by *permanently selecting* one new bull. By considering the dual information from the solution to the last LP we determine which unselected bull b is expected to damage the total profit the least if that bull is selected, and used at least its minimum L_b times. To permanently select bull b we set the lower bound on bull use to L_b in (8.13). This is like setting $\delta_b = 1$ in the original constraint (8.11). We also permanently select any bulls that are currently selected, to guarantee that the solution to the next $\boldsymbol{\delta}$-relaxed CMP model selects (at least) one more bull.

Cowculus is a heuristic solver. It may fail to find a solution (let alone the global maximum) on some feasible problems. For problems where all calves are allowed, so $W_{bc} = 1$ for all matings, then whichever bulls are selected by the end of Step 2 a feasible assignment of cows to these bulls is possible in Step 3. If $W_{bc} \neq 1$ for some matings then it may not be possible to assign cows in Step 3. However, in practice $W_{bc} = 1$ for nearly all calves so this is unlikely to happen, and in fact Cowculus has never been known to fail to find a feasible solution.

Getting reserve bulls

In principle it would be best to consider reserve bulls at the same time as considering the first choice bulls of the standard CMP model. However Cowculus does not attempt to model the need for reserve bulls exactly, but instead uses an approximate method. We decompose the overall problem into three completely separate problems, for 1st, 2nd and 3rd choice bulls, and

solve these three consecutively. This makes the overall problem considerably easier to solve, and avoids having to determine the relative importance of 1st, 2nd and 3rd choice bulls. It is believed that finding the 1st choice bulls is far more important than finding reserve bulls, so little is lost by using this decomposition instead of the original problem. Finding good 1st choice bulls entails solving the standard CMP model, as described above. Finding good reserve bulls is considered here.

When finding 2nd choice bulls we assume that only the subset of bulls selected by the 1st choice solve are available, and that any calves that were produced by the 1st choice solve are vetoed. When selecting 3rd choice bulls we also veto the calves produced by the 2nd choice solve. As it is expected that the selected bulls will already by used plenty of times in the 1st choice solve, and only rarely will the reserve choices be used, we do not impose the condition that in the reserve solves each selected bull b must be selected at least L_b times. This relaxation means a reserve solves may even have better (larger) objective value than a 1st choice solve. Both the reserve solves are considered to be altered $\boldsymbol{\delta}$-relaxed models.

We give three different methods for finding the 2nd (and then the 3rd) choice bulls. In Method 1 we solve the altered $\boldsymbol{\delta}$-relaxed CMP model of using EMSOL. The LP is likely to be fairly easy to solve as the bulls are pre-selected, and several calves have been vetoed. In Method 2 we solve the $\boldsymbol{\delta}$-relaxed CMP model using a regret minimisation algorithm, that awards the popular bulls to the cows where they make the most difference. The regret minimisation algorithm is a fast way to solve the $\boldsymbol{\delta}$-relaxed CMP model, that results in a good approximate solution. In Method 3 we relax the condition that each bull b can be used only U_b times. This results in a problem of type Model 1, which can be solved by simply assigning each cow to the most attractive of the available bulls. The problem is very quick to solve, but may give unacceptable solutions.

	Method 1	Method 2	Method 3
2nd choice bulls	10,879	10,756	11,263
3rd choice bulls	10,601	10,394	10,605

The table above shows the objective value for finding 2nd choice and 3rd choice bulls on problem Prerunex. The 1st choice objective value is 11,134. The three methods vary in speed (not shown) and objective value. As expected Method 3, which allows unlimited use of each bull, has the best objective values. Method 1 and Method 2 solve the same problem but the LP of Method 1 gets better solutions than the heuristic of Method 2. The default method for finding reserve bulls used in Cowculus is Method 1.

8.3.2 Results

Table 8.5 shows results for solving the four test problems with Cowculus, including finding the reserve bulls, which occurs in the final two iterations. For each problem the first iteration

	total time	iterations	number of bulls selected each iteration
Mininew	0.04	4	4, 5, 5, 5
Advrunex	0.09	4	3, 3, 3, 3
Prerunex	2.23	4	7, 8, 8, 8
Prerunex2	2.48	6	5, 6, 7, 8, 8, 8

Table 8.5: Results for Cowculus

usually selects a number of bulls that is close to acceptable, here always equal or below the minimum allowed. For Advrunex the three bulls are pre-selected so in all iterations exactly three bulls are selected. The slowest problem, Prerunex2, still takes well under the suggested time limit of 10 seconds.

In all iterations apart from the first the LP solves are warm started, which saves a lot of time. On average the warm started LPs during the iterations spent finding 1st choice bulls take 51% as many LP iterations as the first LP solve. The warm started LPs spent during iterations to get reserve bulls take 63% as many LP iterations as the first LP solve. The reason the warm start is less effective when finding reserve bulls is that the two LPs solved there are somewhat different problems to the other LPs. Cowculus consists of an LP solver and a few minor subroutines. If the solve is efficient most of the time should be spent solving LPs. The remaining time will be spent either in overheads such as reading in and out of files, or very quick subroutines required by Cowculus, such as counting the number of bulls selected each iteration and comparing it with bounds. For the two small test problems, Mininew and Advrunex, the LP solve is so fast that the overheads dominate the solve time. For the two large problems, Prerunex and Prerunex2, an average of 84% of the time spent solving the problems is spent solving LPs, which is a satisfactory level.

On all four problems Cowculus finds the global maximum for 1st choice bulls. Furthermore, in Horgan [2007] Cowculus was tested on eight different problems, and found the global maximum in all cases. Thus on the twelve known CMP problems tested Cowculus has found the global maximum every time.

Comparison with other solvers

	Cowculus	BONMIN	CBC	CPLEX	FILMINT	MINLP-Fletcher
Mininew	0.04	0.03	0.02	0.01	0.02	0.00
Advrunex	0.07	0.07	0.01	0.02	0.32	0.13
Prerunex	1.33	12.11	5.39	0.52	fail	fail
Prerunex2	1.78	3.98	0.62	0.39	fail	fail

Table 8.6: Standard CMP model solve times

Table 8.6 compares Cowculus to a range of other solvers, all designed to solve ILPs but not designed specifically to solve the cattle mating problem. These other solvers all use branch and bound or branch and cut, so are guaranteed to find the global maximum (as long as they can

solve the problem). All solves are only to find 1st choice bulls for each cow, hence Cowculus is truncated after Step 3 and its solve times are less than those given in Table 8.5. For solvers other than Cowculus the ILP model was built in AMPL and solved via the NEOS server. The two small problems were solved by all the solvers in less than a second. The two large problems were not solved by FILMINT and MINLP-Fletcher, and were solved with varying speed by the other three ILP solvers. The fastest of them was CPLEX, which was faster than Cowculus on all problems.

	Cowculus		CPLEX	
	iterations	LP iterations	B& B branches	LP iterations
Mininew	2	50	0	79
Advrunex	2	513	0	215
Prerunex	2	1073	0	582
Prerunex2	4	1143	0	706

Table 8.7: Results for Cowculus and CPLEX

Table 8.7 directly compares Cowculus and CPLEX, again only for finding the 1st choice bulls. For Cowculus we give the number of iterations and LP iterations, for CPLEX the number of branch and bound branches and LP iterations. For all problems CPLEX requires zero branches hence just one LP solve, so takes few LP iterations. It was hoped that the quasi-random calf scores in Prerunex2 would make a branch and bound solve difficult and so favour a heuristic solver like Cowculus. CPLEX still solved Prerunex2 quickly though. Cattle mating problems that are much larger than these test problems will not be significantly harder when solved by Cowculus, as LP scales well with problem size, but may become significantly harder when solved by CPLEX, as branch and bound ILP does not scale so well with problem size. Conversely, there may exist problems for which the ILP solvers find the global maximum and Cowculus does not.

In summary although it is not as fast as CPLEX, Cowculus solves the test problems accurately, finding the global minimum on all test problems, and quickly, solving well within 10 seconds.

Cowculus is currently used at the ICBF, implemented by John McCarthy. As of May 2008 the ICBF have used Cowculus for about 36,000 solves for over 7,000 farmers, with no problems with the solver.

8.4 Summary

In this chapter we introduced a type of assignment problem called the cattle mating problem. We described different versions of it, including an ILP known as the standard CMP model. The solver Cowculus was developed to solve this model. It relaxes one set of integer variables to give a relaxed problem, which, though still an ILP, can be solved by LP. A series of these

LPs are solved while altering the value of the relaxed integer variables, until an integer feasible solution is found. Finally Cowculus heuristically finds reserve bulls for each cow.

Cowculus solves the four test problems described here, and eight other test problems, finding the global maximum in all cases. The solve exploits warm starts and is fairly efficient, solving all the problems in well under ten seconds. It is competitive with some ILP solvers but slower than CPLEX. Cowculus is currently in use at the ICBF.

8.4.1 Further work

In Section 8.1 we mentioned that local improvement methods are often used to refine an approximate solution found by a heuristic solver. Local improvement methods could be added to Cowculus. On all the test problems Cowculus currently finds the global maximum already, but these methods could be useful for harder problems.

If the ICBF are resolving different versions of the same problem, which only differ in objective costs, it may be possible to use the optimal LP basis of a previous solution to warm start a new problem. For some problems, such as the advanced CMP model Advrunex, it may also be possible to detect that all bulls must be selected and solve the problem as an LP.

There are several possible extensions to the standard CMP model that were discussed in consultation with the ICBF. These could be implemented in a future version of Cowculus. They include; equalising expected milk yield, catering for heifers, a new approach for factoring in the effects of inbreeding and heterosis and retaining high value calves.

Equalising expected milk yield

It may be convenient for the farmer to have a herd where all cows produce about the same yield of milk. Depending on the farmer this may simply mean the cows produce the same milk yield as each other, or the same milk yield as a given constant. In either case if the expected milk yield of potential calves is known this extension can easily be modelled with the addition of a few extra linear constraints to the model.

Catering for heifers

Certain bulls are known to produce large calves, which are unsuitable to mate with heifers (cows who have not given birth before). There are various methods considered to deal with this.

Firstly we can simply veto matings between a bull b expected to produce large calves with a heifer c, by setting $W_{bc} = 0$. If we only want to discourage production of a calf, but not veto it, this can be done by giving the mating a low score S_{bc}. Secondly we could force Cowculus to select some *easy calving bulls*, which are suitable to mate with heifers, even if this does not appear optimal. Thirdly the farmer could split the cows into two subherds, with one

subherd consisting of only heifers and easy calving bulls. Each herd could then be assigned bulls separately.

Modelling inbreeding and heterosis

Two contributors to the score of each calf are the amount of inbreeding and the amount of heterosis between the parents. If inbreeding and heterosis levels for each calf are known the calf scores may be adjusted accordingly, such as in (8.15). However, finding the inbreeding and heterosis levels for the calves is a time consuming process, as it involves looking up the ancestors of every calf in a database, and for a typical problem such as Prerunex there are $BC = 23540$ potential calves.

We now suggest an approximate method to model inbreeding and heterosis, without having to find the inbreeding and heterosis levels of all calves. This could be implemented with Cowculus, though has not been as far as is known. In the first step the model is solved with simple calf scores involving no inbreeding and heterosis terms. In the solution to this solve some bulls will be selected, typically considerably less than the total number of bulls. In the second step we calculate revised calf scores for all potential calves parented by the selected bulls, including inbreeding and heterosis terms. In the third step we resolve, pre-selecting those bulls selected by the first solve. Providing that the bulls selected by the first solve remain among the best bulls once inbreeding and heterosis levels are factored in, this method will give a good solution to the original problem.

Retaining high value calves

Farmers tend to sell some proportion of their calves each year, and keep the rest in the herd. The retained calves will breed and be the ancestors to future generations. There is thus an incentive to make these calves of very high value. This can be modelled by introducing extra integer variables to determine if a calf is to be retained or not, and altering the objective function to prioritise selecting retained calves of high value. These extra integer variables could make the problem much harder to solve.

Appendix A

Hyper-sparse and Dense-fill problems

		columns	rows	density of initial tableau	density of optimal tableau
0	STORMG28	10193	4409	0.61022E-03	0.24464E-02
1	80BAU3B	9799	2262	0.94752E-03	0.49392E-01
2	STOCFOR2	2031	2157	0.19044E-02	0.61413E-01
3	SCTAP3	2480	1480	0.24177E-02	0.72997E-02
4	SHIP12S	2763	1151	0.25715E-02	0.66216E-02
5	SHIP12L	5427	1151	0.25887E-02	0.59922E-02
6	SIERRA	2036	1227	0.29229E-02	0.11374E-01
7	GANGES	1681	1309	0.31412E-02	0.26060E-01
8	CZPROB	3523	929	0.32598E-02	0.15489E-01
9	SCTAP2	1880	1090	0.32764E-02	0.13300E-01
10	GFRD-PNC	1092	616	0.35337E-02	0.27196E-01
11	SHELL	1775	536	0.37376E-02	0.35844E-01
12	CYCLE	2857	1903	0.38110E-02	0.91724E-01
13	SHIP08S	2387	778	0.38307E-02	0.83889E-02
14	SHIP08L	4283	778	0.38419E-02	0.85344E-02
15	WOODW	8405	1098	0.40606E-02	0.10080
16	SCRS8	1169	490	0.55551E-02	0.12500
17	SCFXM3	1371	990	0.57298E-02	0.68367E-01
18	STANDMPS	1075	467	0.73283E-02	0.31929E-01
19	STANDGUB	1184	361	0.73440E-02	0.94730E-02
20	SHIP04S	1458	402	0.74252E-02	0.13939E-01
21	SHIP04L	2118	402	0.74368E-02	0.13695E-01
22	FINNIS	614	497	0.75698E-02	0.32914E-01
23	STANDATA	1075	359	0.78539E-02	0.10212E-01
24	MAROS	1443	846	0.78753E-02	0.53965
25	SEBA	1028	515	0.82203E-02	0.94141E-02
26	SCFXM2	914	660	0.85919E-02	0.11784
27	ETAMACRO	688	400	0.87536E-02	0.16055
28	FIT1P	1677	627	0.93849E-02	0.28293E-01
29	SCORPION	358	388	0.10266E-01	0.58652E-01
30	SCTAP1	480	300	0.11750E-01	0.46931E-01
31	FFFFF800	854	524	0.13915E-01	0.66423E-01
32	SCFXM1	457	330	0.17167E-01	0.11175
33	CAPRI	353	271	0.18471E-01	0.25826
34	VTP.BASE	203	198	0.22590E-01	0.21548
35	LOTFI	308	153	0.22876E-01	0.76755E-01
36	TUFF	587	333	0.23124E-01	0.20940
37	BOEING1	384	351	0.25856E-01	0.15486
38	AGG2	302	516	0.27491E-01	0.92632E-01
39	AGG3	302	516	0.27594E-01	0.95693E-01
40	AGG	163	488	0.30298E-01	0.15058
41	STOCFOR1	111	117	0.34419E-01	0.37253
42	RECIPE	180	91	0.40476E-01	0.43223E-01
43	BOEING2	143	166	0.50383E-01	0.12731
44	BEACONFD	262	173	0.74461E-01	0.91404E-01

Table A.1: Size and density of Hyper-sparse problems

		columns	rows	density of initial tableau	density of optimal tableau
45	SCAGR25	500	471	0.65987E-02	0.18161
46	BNL1	1175	643	0.67781E-02	0.58834
47	NESM	2923	662	0.68671E-02	0.21784
48	SCSD8	2750	397	0.78626E-02	0.58377
49	PILOT4	1000	410	0.12539E-01	0.80783
50	SC205	203	205	0.13240E-01	0.90191
51	DEGEN2	534	444	0.16778E-01	0.17264
52	BANDM	472	305	0.17324E-01	0.54256
53	BORE3D	315	233	0.19470E-01	0.28159
54	SCSD6	1350	147	0.21749E-01	0.18424
55	STAIR	467	356	0.23194E-01	0.93127
56	SCAGR7	140	129	0.23256E-01	0.20194
57	SC105	103	105	0.25890E-01	0.83403
58	BRANDY	249	220	0.39211E-01	0.55511
59	SCSD1	760	77	0.40807E-01	0.22779
60	E226	282	223	0.40995E-01	0.46416
61	SHARE1B	225	117	0.43723E-01	0.21918
62	SC50B	48	50	0.49167E-01	0.96000
63	SC50A	48	50	0.54167E-01	0.86917
64	GROW7	301	140	0.61984E-01	0.71592
65	ADLITTLE	97	56	0.70508E-01	0.36690
66	BLEND	83	74	0.79941E-01	0.40687
67	SHARE2B	79	96	0.91508E-01	0.39214
68	ISRAEL	142	174	0.91833E-01	0.71333
69	AFIRO	32	27	0.96065E-01	0.17477
70	KB2	41	43	0.16222	0.75156
71	FIT2D	10500	25	0.49150	0.99485
72	FIT1D	1026	24	0.54435	0.85551

Table A.2: Size and density of Densefill problems

Appendix B

Estimating final tableau density

In Chapter 2 we stated formula (2.4) for the link between the number of structurals in the basis, k, and the average number of nonzeros in each column at iteration k, a_k. m and n are the number of rows and columns of the initial tableau respectively, and C_1, C_2 and C_3 are constants. This formula assumes that the nonzeros in the tableau are distributed randomly each iteration.

$$k = C_1 \log\left(\frac{a_k - n}{a_0 - n}\right) + C_2 \log\left(\frac{a_k - 1}{a_0 - 1}\right) + C_3 \log\left(\frac{a_k n - m}{a_0 n - m}\right). \tag{B.1}$$

We now derive this formula, and give explicit values for the three constants. Firstly the Markovitz count each iteration is the product of the nonzeros in the pivot column, $(a_k - 1)$, and the nonzeros in the pivot row, $(\frac{a_k m}{n} - 1)$. The proportion of nonzeros in the tableau is $(1 - \frac{a_k}{n})$. The increase in nonzeros per column each iteration is the Markovitz count, multiplied by the proportion of nonzeros in the tableau, to account for superposition, divided by the number of columns. Writing this difference in nonzeros as $\frac{da_k}{dk}$ we get (B.2). This is separated into partial fractions in (B.3) and the integration carried out in (B.4), yielding a constant C_0. In (B.5) we use the fact that at $k = 0$ $a_k = a_0$ to solve for C_0.

$$\frac{da_k}{dk} \simeq \frac{(a_k - 1)(\frac{a_k m}{n} - 1)(1 - \frac{a_k}{n})}{m} \tag{B.2}$$

$$= -\frac{mn}{(a-n)(n-1)(m-1)} - \frac{mn^2}{(a-1)(n-m)(n-1)} + \frac{m^3 n}{(am-n)(n-m)(m-1)} \tag{B.3}$$

$$k = -\frac{mn\log(a_k - n)}{(n-1)(m-1)} - \frac{mn^2\log(a_k - 1)}{(n-m)(n-1)} + \frac{m^2 n\log(a_k m - n)}{(n-m)(m-1)} + C_0 \tag{B.4}$$

$$= -\frac{mn\log(\frac{a_k - n}{a_0 - n})}{(n-1)(m-1)} - \frac{mn^2\log(\frac{a_k - 1}{a_0 - 1})}{(n-m)(n-1)} + \frac{m^2 n\log(\frac{a_k m - n}{a_0 m - n})}{(n-m)(m-1)} \tag{B.5}$$

We have now derived (B.1), with the constants defined as

$$C_1 = -\frac{mn}{(n-1)(m-1)}, \quad C_2 = -\frac{mn^2}{(n-m)(n-1)}, \quad C_3 = \frac{m^2 n}{(n-m)(m-1)}$$

Appendix C

Estimating number of nonzeros in rows and columns

In Section 3.4.4 we considered how to estimate the number of nonzeros in each rows and columns after several iterations of the revised simplex method. We do this by first producing a minimal list of pivots that take the tableau from the initial tableau to the current basis. We call these the *Tarjan pivots*. The Tarjan pivots each add a structural to the basis, so are expected to cause fill in in any rows and columns that intersect the pivot column and row.

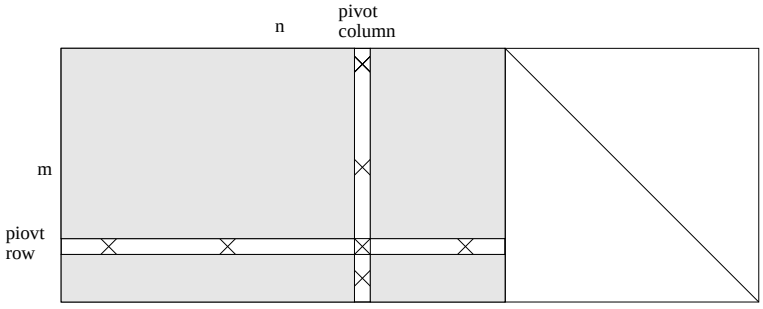


Figure C.1: Example tableau with pivot row and column marked

Let m and n be the number of rows and columns of the initial tableau respectively, and K_1 and K_2 be constants to account for numerical and structural cancellation (so they are typically less than one). Let e_i^r represents the estimated row density of each of the $i \in I$ rows affected by the pivot, and e_j^c be the estimated column number of nonzeros in each of the $j \in J$ columns affected by the pivot. Then by considering Figure C.1, and using a similar line of reasoning to that in Section B, formulas for updating both the row and column density can be derived in (C.1) and (C.2).

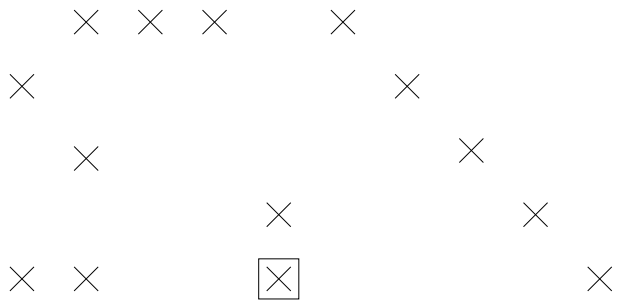
$$e_i^r = e_i^r + K_1 K_2 (|J| - 2) \frac{(n - e_i^r + 1)}{(n-1)}, \tag{C.1}$$

$$e_j^c = e_j^c + K_1 K_2 (|I| - 1) \frac{(m - e_i^c)}{(m-1)}. \tag{C.2}$$

$$e_{j_2}^c = e_{j_1}^c, \tag{C.3}$$

$$e_{j_1}^c = 1. \tag{C.4}$$

$|I|$ and $|J|$ are the number of nonzeros in the pivot column and row respectively. The reason that the row and columns formulas differ slightly is that we imagine the identity matrix explicitly included in the tableau, which means each row has an extra nonzero in it. Furthermore we must consider two special columns; let j_1 be the column entering the basis (so will become an identity column), and j_2 be the column leaving the basis (so will inherit the old number of nonzeros of j_1). This gives (C.3) and (C.4).



We now demonstrate the formula on the example tableau above. Note the identity matrix is included. Here $m = 5$ and $n = 4$. Suppose the pivot is at the element marked with a square, so $I = \{1, 4, 5\}$ and $J = \{1, 2, 4, 9\}$. Then applying (C.1) to row one and assuming no numerical cancellation ($K_1 = 1$) and a standard amount of superposition ($K_2 = 1$) we get that:

$$\begin{aligned} e_1^r &= e_1^r + (|J| - 2) \frac{(m - e_1^r + 1)}{(m-1)}, \\ &= 4 + (4-2) \frac{(4 - 4 + 1)}{(4-1)}, \\ &= 4 + \frac{2}{3}. \end{aligned}$$

Thus we expect the pivot to cause an extra $\frac{2}{3}$ nonzeros in the first row. This can also be seen by considering that there is one empty space in the first row, out of a maximum of three possible empty spaces, and two extra nonzeros from the pivot row are going to be added. There is thus a $\frac{2}{3}$ chance of fill in (you can see that here there is in fact going to be exactly one element of fill in in the first row)

Similarly for the first column:

$$
\begin{aligned}
e_1^c &= e_1^c + 1(|I| - 2)\frac{(n - e_1^c)}{(n-1)}, \\
&= 2 + (3-1)\frac{(5-2)}{(5-1)}, \\
&= 2 + \frac{3}{2},
\end{aligned}
$$

so we expect on average 1.5 extra nonzeros in the first column after the pivot (here there is in fact going to be exactly two elements of fill in in the first column).

Bibliography

M.A. Abramson. *Pattern Search Algorithms for Mixed Variable General Constrained Optimization Problems*. PhD thesis, Rice University, 2002.

H. Adhya and M. Tawarmalani. A lagrangian approach to the pooling problem. *Ind. Eng. Chem. Res*, 1999.

Y. Agarwal, K. Mathur, and H.M. Salkin. A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks*, 1989.

H. Almutairi. *A Lagrangean Relaxation and A Heuristic for the Pooling Problem*. PhD thesis, University of Waterloo, 2008.

M. Amini and M. Racer. A hybrid heuristic for the generalized assignment problem. *European Journal of Operational Research*, 1995.

F. Amos and M. Ronnqvist. Modelling the pooling problem at the new zealand refinery company. *Journal of Operational Research Society*, 1997.

F. Archetti and F. Schoen. A survey of the global and optimization problem: General theory and computational approaches. *Annals of Operations Research*, 1984.

C. Audet, J. Brimberg, P. Hansen, S. Le Digabel, and N. Maldenovic. Pooling problem: Alternate formulations and solution methods. *Management Science*, 2004.

T.E. Baker and L.S. Lasdon. Successive linear programming at exxon. *Management Science*, 1985.

P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Waechter. Branching and bounds tightening techniques for non-convex minlp. Technical report, IBM Research Report, 2008.

H. Ben-Amor and J. Desrosiers. A proximal trust-region algorithm for column generation stabilization. *Computers and Operations Research*, 2006.

A. Ben-Tal, G. Eiger, and V. Greshovitz. Global minimization by reducing the duality gap. *Mathematical Programming*, 1994.

J. F. Benders. Partitioning procedures for solving mixed-variables programming problems,. *Numer. Math.*, 1962.

Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, 1982.

C.G.E. Boender and A.H.G. Rinnooy Kan. A bayesian analysis of the number of cells of a multinomial distribution,. *Statistician*, 1983.

H.P.J. Bolton, J.F. Schutte, and A.A. Groenwold. Multiple parallel local searches in global optimization. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000.

S. Boyd and L. Vandenberghe. Localization and cutting-plane methods. Technical report, Stanford University, 2003.

A.L. Brearly, G. Mitra, and H.P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 1975.

Choong Ming Chin and Roger Fletcher. On the global convergence of an SLP-filter algorithm that takes EQP steps, 2003.

G. Cohen and D. Li Zhu. Decomposition coordination methods in large scale optimization problems: The nondifferentiable case and the use of augmented lagrangians. *Adv. Large Scale Syst.*, 1984.

R. Cohen, L. Katzir, and D. Raz. An efficient approximation for the generalized assignment problem. *Information Processing Letters*, 2006.

Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region methods*. SIAM, 2000.

George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 1960.

I.S. Duff. On permutations to block triangular form. *J. Inst. Math. Appl.*, 1977.

I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

I. El-Samahy. A heuristic method to reactive power service procurement. *IEEE MELECON*, 2006.

T.H. Emigh. On the number of observed classes from a multinomial distribution. *Biometrics*, 1983.

W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 1968.

M.C. Ferris and J.D. Horn. Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 1998.

M. Fieldhouse. The pooling problem. in optimization in industry. *John Wiley and Sons*, 1993.

M. Fischetti and A. Lodi. Repairing mip infeasibility through local branching. *University of Padova*, 2005.

M. Fischetti, F.Glover, and A. Lodi. The feasibility pump. *Math. Program.*, 2005.

Roger Fletcher. *Practical Methods of Optimization*. Wiley, 1981.

Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. Technical report, University of Dundee, 1997.

Roger Fletcher and Sven Leyffer. *User manual for filterSQP*, 1999.

C.A. Floudas and A. Aggrawal. Global optimum search for nonconvex nlp and minlp problems. *Comput.Chem.Eng*, 1990.

Format International. http://www.formatinternational.com/, 2009.

L.R. Foulds and J.M. Wilson. A variation of the generalized assignment problem arising in the new zealand dairy industry. *Annals of Operations Research*, 1997.

L.R. Foulds, D. Haugland;, and K.A. Jornsten. A bilinear approach to the pooling problem. *Optimization*, 1992.

R. Fourer. Solving staircase linear programs by the simplex method. *Mathematical Programming*, 1983.

A. Frangioni. Generalized bundle methods. *SIAM J. Optim.*, 2002.

A.P. French and J.M. Wilson. An lp-based heuristic procedure for the generalised assignment problem with special ordered sets. *Computers and Operations Research*, 2007.

S.G. Garille and S.I. Gass. Stigler's diet problem revisited. *Oper. Res.*, 2001.

A.M. Geoffrion. Generalized benders decomposition. *Journal of Optimization Theory and Applications*, 1972.

S.T. Glad. Properties of updating methods for the multipliers in augmented lagrangian. *Journal of Optimization Theory and Applications*, 1979.

I.J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 1953.

R.E. Griffith and R.A. Stewart. A nonlinear programming technique for the optimization of continuous processing systems. *Management Sci.*, 1961.

A. Grothey. *Decomposition Methods for Nonlinear Nonconvex Problems*. PhD thesis, University of Edinburgh, 2001.

A. Grothey, R. Fawcett, and K.I.M. McKinnon. Solving a feed-mix pooling problem in agriculture. *OR41 Conference Proceedings*, 1999.

V.R. Guevara. Use of nonlinear programming to optimize performance response to energy density in broiler feed formulation. *Poultry Science*, 2004.

Eldon A. Gunn. A decomposition method based on the augmented lagrangian. *INFOR*, 1988.

J.A.J. Hall. Emsol, 1997.

J.A.J. Hall and K.I.M. McKinnon. Hyper-sparsity in the revised simplex method and how to exploit it. *Comput. Optim. Appl.*, 2005.

C.A. Haverly. Studies of the behavior of recursion for the pooling problem. *ACM SIGMAP Bulletin*, 1978.

G. Hertzler, D.E. Wilson, D.D. Loy, and G.H. Rouse. Optimal beef cattle diets formulated by nonlinear programming. *Animal Science*, 1988.

M.R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 1969.

W. Horgan. MSc Thesis, 2007. University of Edinburgh.

X.X. Huang and Y.Q. Yang. A unified augmented lagrangian approach to duality and exact penalization. *Mathematics of Operations Research*, 2003.

ICBF. http://www.icbf.com/, 2006.

B. Kallehauge, J. Larsen, and O.B.G. Madsen. Lagangean duality applied on vehicle routing with time windows. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, 2001. Technical report IMM-TR-2001-9.

Markku Kallio and Evan L. Porteus. Triangular factorization and generalized upper bounding techniques. *Operations Research*, 1977.

Spyridon Kontogiorgis and Robert R. Meyer. A variable-penalty alternating directions method for convex optimization. *Mathematical Programming: Series A and B archive*, 1998.

T. Larsson and M. Patriksson. An augmented lagrangian dual algorithm for link capacity side constrained traffic assignment problems. *Transportation Research B*, 1995.

T. Larsson and D. Yuan. An augmented lagrangian algorithm for large scale multicommodity routing. *Computational Optimization and Applications*, 1994.

L.S. Lasdon and A.D Waren. A survey of nonlinear programming applications. *Operations Research*, 1980.

J. Lee. Mixed-integer nonlinear programming: Some modelling and solution issues. *IMB J. Res and Dev.*, 2007.

Claude Lemarechal. Lagrangian relaxation. In *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions*. Springer-Verlag, 2001.

M. Lentini, A. Reinoza, A. Teruel, and A. Guillen. Simpar: A parallel sparse simplex. *Comp. Appl. Math.*, 1995.

A.N. Letchford. Mixed-integer non-linear programming: A survey. Slides, 2009. Lancaster University Management School.

S. Leyffer and J. Linderoth. Mixed integer nonlinear programming. SIAM Conference on Optimization, 2005.

L. Liberti and C.C Pantelides. An exact reformulation algorithm for large nonconvex nlps involving bilinear terms. *Journal of Global Optimization*, 2006.

C. Lim and H.D. Sherali. Convergence and computational analyses for some variable target value and subgradient deflection methods. *Comput Optim Appl*, 2006.

M. Locatelli and F. Schoen. Random linkage: a family of acceptance/rejection algorithms for global optimization. *Mathematical Programming*, 1999.

O.L. Mangasarian. Unconstrained lagrangians in nonlinear programming. *SIAM J. Control*, pages 772–791, 1975.

F. Margot. Symmetry in integer linear programming. Technical report, Carnegie Mellon University, 2008.

R.E. Marsten. The use of the boxstep method in discrete optimization. *Math. Programming Stud.*, 1975.

Richard D. McBride. Progress made in solving the multicommodity flow problem. *SIAM J. on Optimization*, 1998.

G.P. McCormick. *Nonlinear programming: theory, algorithms and applications*. John Wiley and sons, 1983.

S. McParland, J.F. Kearney, M. Rath, and D.P. Berry. Inbreeding trends and pedigree analysis of irish dairy and beef cattle populations. *J. Anim. Sci.*, 2007a.

S. McParland, J.F. Kearney, M. Rath, and D.P. Berry. Inbreeding effects on milk production, calving performance, fertility, and conformation in irish holstein-fresians. *J. Dairy. Sci.*, 2007b.

C. Meyer and C. Floudas. Global optimization of a combinatorially complex generalised pooling problem. *AIChE Journal*, 2006.

A. Miele, E.E. Cragg, and A.V. Levy. Use of the augmented penalty function in mathematical programming problems: Part 1. *Journal of Optimization Theory and Applications*, 1971.

Alan G. Munford. The use of iterative linear programming in practical applications of animal diet formulation. *Math. Comput. Simul.*, 1996.

D.L. Nash and G.W. Rogers. Risk management in herd sire portfolio selection: A comparison of rounded quadratic and separable convex programming. *J. Dairy Sci.*, 1996.

Tatsushi Nishi, Masakazu Ando, and Masami Konishi. A distributed route planning method for multiple mobile robots using lagrangian decomposition and coordination techniques. *IEEE Transactions on Robotics*, 2005.

Ivo Nowak and Stefan Vigerske. Lago - a (heuristic) branch and cut algorithm for nonconvex minlps. *Central European Journal of Operations Research*, 2007.

I.H. Osman. Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches. *OR Spektrum*, 1975.

Bruce W. Patty. The basis suppression method. *Annals of Operations Research*, 1989.

M.J.D. Powell. A method for nonlinear constraints in minimization problems. Optimization, Academic Press, pp 283-298, 1969.

I. Quesada and I.E; Grossman. Global optimization of bilinear process networks and multicomponent and multicomponent flows. *Comput.Chem.Eng*, 1995.

A.H.G. Rinnooy Kan and C.G.E. Boender. A multinomial bayesian approach to the estimation of population and vocabulary size. *Biometrika*, 1987.

B.D. Ripley. *Statistical Inference for Spatial Processes*. Cambridge University Press, 1988.

Robert Fourer Robert E. Bixby. Finding embedded network rows in linear programs i. extraction heuristics. *Management Science*, 1988.

R.T. Rockafellar. Augmented lagrangian multiplier functions and duality in nonconvex programming. *SIAM J. Control*, 1974.

R.T. Rockafellar. Augmented lagrangians and applications of the proximal point algorithm in convex programming. *Mathematics of Operational Research*, 1976.

R.T. Rockafellar and Roger J.B. Wets. *Variational Analysis*. Springer, 1998.

G.T. Ross and R.M. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming*, 1975.

R. Rotondi and S. Drappo. A clustering method for global optimization based on the kth nearest neighbour. *Statistics and Computing*, 1995.

A. Ruszczynski. On convergence of an augmented lagrangian decomposition method for sparse convex optimization. *Mathematics of Operational Research*, 1995.

M. Saunders. *Sparse Matrix Computations: A fast, stable implementation of the simplex method using Bartels-Golub updating*. Academic Press, 1976.

H.D. Sherali and A. Alameddine. A new reformulation-linearization technique for bilinear programming problems. *Journal of Global Optimization*, 1992.

H.D. Sherali and J.C. Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 2001.

J.D. Simon and H.M. Azma. Exxon experience with large scale linear and nonlinear programming applications. *Comput.Chem.Eng*, 1983.

Sloten International. http://www.sloten.com/, 2009.

J.A. Snyman and L.P. Fatti. A multi-start global minimization algorithm with dynamic search trajectories. *J. Optim. Theory Appl.*, 1987.

G. Stigler. The cost of subsistence. *Journal of Farm Economics*, 1945.

J.R Stokes and R. Tozer. Optimal feed mill blending. *Review of Agricultural Economics*, 2006.

X.L. Sun, D. Li, and K.I.M. McKinnon. On saddle points of augmented lagrangians for constrained nonconvex optimization. *SIAM J. on Optimization*, 2005.

R.A. Tapia. Diagonalized multiplier methods and quasi-newton methods for constrained optimization. *Journal of Optimization Theory and Applications*, 1977.

M. Tawarmalani and N. Sahinidis. Convexification and global optimization in continuous and nonlinear mixed-integer nonlinear programming: theory, algorithms, software, and applications. *Kluwer*, 2002.

Tamas Terlaky and Shuzhong Zhang. Pivot rules for linear programming: A survey on recent theoretical developments. *Annals of Operation Research*, 1993.

Tran Vu Thieu. A note on the solution of bilinear programming problems by reduction to concave minimization. *Mathematical Programming*, 1986.

A. Torn. Probabilistic global optimization, a cluster analysis approach. *Proc. 2nd European Congress on Operations Research*, 1976.

M.A. Trick. A linear relaxation heuristic for the generalised assignment problem. *Naval Research Logistics*, 1992.

S.S. Tripathi and K.S. Narrendra. Constrained optimization problems using multiplier methods. *Journal of Optimization Theory and Applications*, 1972.

Zsolt Ugray, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Martí. Scatter search and local nlp solvers: A multistart framework for global optimization. *INFORMS J. on Computing*, 2007.

G.N. Vanderplaats. *Numerical optimization techniques for engineering design*. McGrawHill, 1984.

V. Visweswaran and C.A Floudas. New properties and computational improvement of the gop algorithm for problems with quadratic objective functions and constraints. *J. Global Optimiz.*, 1993.

T.C. Wagner. Defining properties for decomposition in nonlinear programming. Technical report, Michigan College of Engineering, 1997.

P. Wentges. Weighted dantzig-wolfe decomposition of linear mixed-integer programming. *Optimization and Engineering*, 1997.

H. P. Williams and A. C. Redwood. A structured linear programming model in the food industry. *Operational Research Quarterly*, 1974.

J.M. Wilson. An algorithm for the generalized assignment problem with special ordered sets. *Journal of Heuristics*, 2005.

Wayne L. Winston. *Introduction to Mathematical Programming*. Duxbury, 2000.

M. Yuceer and R. Berber. A heuristic approach for solution of minlp problem for production scheduling in preemptive mode. *Chemical Engineering and Processing*, 2004.